A Multiple Integrated Consensus Protocol based on Paxos, FastPaxos and Fast Paxos

Michel Hurfin, Izabela Moise

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1941 – December 2009

A Multiple Integrated Consensus Protocol based on Paxos, FastPaxos and Fast Paxos

Michel Hurfin[*] , Izabela Moise[**]
Michel.Hurfin@inria.fr, Izabela.Moise@irisa.fr

Abstract: We consider an asynchronous distributed system prone to crash failures and present a protocol designed to solve several consecutive consensus instances. After specifying the Multiple Integrated Consensus problem, we propose a solution that follows the Paxos approach, but relies on another flexible interaction scheme. A subset of processes (namely the coordinators and the acceptors) ensures that eventually a single value is selected to become the decision value. Moreover, these processes also act to guarantee the persistence of the previous decisions and to regulate the sequence of consensus instances. In a recent past, two different protocols, namely FastPaxos by Boichat *et al.* and Fast Paxos (with a space) by Lamport, have been designed to reduce the latency of learning a decision value to respectively, three and two communication steps, in favorable circumstances. Our protocol unifies these two different strategies, in order to obtain the best performance gain, in some frequent scenarios.

Key-words: Fault tolerance, Agreement, Consensus, Paxos, Distributed reliable algorithms

---

Un protocole pour des consensus multiples intégrés fondé sur Paxos, FastPaxos et Fast Paxos

Résumé : *Nous considérons un système réparti asynchrone, susceptible de connaître des défaillances de type panne franche et nous présentons un protocole conçu pour résoudre plusieurs instances consécutives de consensus. Après avoir spécifier le problème des Consensus Multiples Intégrés, nous proposons une solution qui suit l'approche Paxos mais qui s'appuie sur un autre schéma d'interaction flexible. Un sous-ensemble de processus (à savoir les coordinateurs et les accepteurs) assurent qu'une seule valeur est finalement sélectionée pour devenir la valeur de décision. De plus, ces processus agissent également pour garantir la persistence des décisions précédentes et pour réguler la séquence d'instances de consensus. Dans un passé récent, deux protocoles différents, à savoir, FastPaxos proposé par Boichat et al. et Fast Paxos (avec un espace) proposé par Lamport, ont été conçus pour réduire la latence lors d'une prise de décision à respectivement trois et deux étapes de communication lorsque les circonstances sont favorables. Notre protocole unifie ces deux différentes stratégies afin d'obtenir le meilleur gain de performance dans des scénarios fréquents.*

Mots clés : *Tolérance aux défaillances, Problème d'accord, Consensus, Paxos, Algorithmes répartis fiables*

---

[*] INRIA Rennes Bretagne Atlantique
[**] IRISA - Université de Rennes 1

1 Introduction

In an asynchronous distributed system prone to crash failures, providing efficient solutions to agreement problems is a key issue when designing fault-tolerant applications. The state machine approach [14] illustrates this concern. In this particular example, replicas of a critical server need to agree on a sequence of incoming requests. Such a sequence is usually constructed by repeatedly calling a Consensus service. The classical specification of the Consensus problem [12] requires that each participant proposes an initial value during an invocation of the *Propose* primitive and, despite failures, all the correct processes have to decide on a single value selected out of these proposed values. In a pure asynchronous system, this problem is impossible to solve [5]. Yet under some well-identified additional synchrony properties which can be indirectly exploited by a failure detector or a leader election service, several consensus protocols have been proposed. Within this paper, we refer to some $\Diamond$S-based consensus protocols [2, 3] but we focus mainly on Paxos [8, 10] and some Paxos-like algorithms [1, 9, 13]. All these deterministic algorithms require a majority of correct participants.

The Paxos protocol has been presented by Lamport first in [7] and described later in a simpler way in [8]. Lamport has identified four basic roles: proposer, learner, coordinator, and acceptor. Each participant may take on multiple roles or just a single one. Proposers are entities that may provide initial values. During a consensus instance, it is assumed that at least one non-crashed proposer must supply an input. The learners are in charge of detecting that the protocol has successfully converged toward a decision value. If f is the maximal number of failures that may occur, at least $f+1$ coordinators, $f+1$ learners, and $2f+1$ acceptors should be defined. Herein, we assume that the protocol is executed by n processes with $f < n/2$. Proposers and learners are not involved in the convergence procedure which is only driven by the interactions between coordinators and acceptors. Coordinators and acceptors play a central role in ensuring that eventually a single value is selected to become the decision value. A leader election service is used to grant eventually a privilege to a single coordinator. If a correct coordinator becomes the unique leader forever (or at least, till the current consensus instance ends), it is able to impose a selected value to a majority of acceptors and to detect the successful termination of its attempt. Acceptors are used to implement quorums as majority sets. Therefore, by assumption, a majority of acceptors should never crash during the computation. In Paxos, a participant to a consensus instance is neither required to invoke the *Propose* primitive with an initial value nor to wait for the returned decision value. Due to the splitting into several roles, the designer breaks free from the classical rigid interaction scheme. A rather stable subset of processes, namely the coordinators and acceptors, takes on the responsibility for defining a unique sequence of decisions.

Within this paper, we revisit the interaction scheme between proposers, learners, coordinators, and acceptors. We formally define the Multiple Integrated Consensus problem and consider a protocol in charge of the whole sequence of consensus instances. Consensus instances are still executed sequentially but not in a complete isolation from each other. We extend the remit of the sub-group of coordinators and acceptors so that they also have to ensure the availability of the past decisions and they have to control when a new consensus instance can start. In the context of a long lasting computation performed by a (potentially large) collection of (possibly ephemeral) processes, the core of dedicated processes formed by the coordinators and acceptors is able, on one hand, to provide all the decision values already computed (or only the most recent ones) to any current member of the collection and, on the other hand, to ensure the progress of the successive consensus instances while regulating the activity of the proposers that may dynamically join and leave the collection. By definition, the k^{th} decision value corresponds to the outcome of the k^{th} consensus instance which selects an initial value v, proposed by at least one member and generates a decision $< v, k >$. A member of the collection may ignore this outcome but, instead of this decision, it cannot consider another couple $< v', k >$ with $v' \neq v$. To regulate the rate of consensus instances, a classical constraint is used: a participant is not allowed to act as a proposer during consensus instance k if it is not able to access the $k-1$ previous decisions. Due to this restriction, whenever it is necessary, a proposer can take into account the past decisions before computing a new initial value. A solution to the total order broadcast problem, which relies on this property, was proposed in [3]: a message already ordered in a previous decision is never proposed again. At the application level, consensus instances seem to be executed sequentially[1].

In a recent past, we have studied the interest of a consensus-based approach in two distinct application domains related to intrusion detection [6] and Grid [11]. In both developments, the repeated and intensive use of a consensus building block militates in favor of an optimization of the performance of this basic agreement protocol. Regarding the Paxos algorithm, two main strategies have already been proposed in a recent past, namely FastPaxos (without space) described in [1] and Fast Paxos (with a blank) presented in [9].

The first strategy, FastPaxos, tries to benefit from the stability of an elected leader during long lasting failure-free synchronous periods. A gain can be observed if circumstances are favorable during two consecutive consensus instances. In [2] (with a $\Diamond$S-based consensus protocol) and in [1] (with a Paxos-like protocol), the authors suggest to keep, even after the end of a consensus instance, the identity of the coordinator that has made the last decision and to reuse this information during the next consensus instance. In [1], when a (potentially new) leader starts the next consensus instance, this information can be

[1] However, this constraint has been relaxed in a few works. For instance, in [1], some consensus instances may run in parallel and provide their respective decisions in an unpredictable order.

exploited to optimize the decision latency. Indeed, if the leader has not changed in between, the first phase required in the original Paxos protocol (called the *Prepare* phase), is useless and, in favorable circumstances, the new consensus instance just requires three communication steps. The principle used to reduce the number of communication steps in FastPaxos is also adopted in other works: for example, by Lampson in [10] (where the notion of view is proposed) and in the work of Martin and Alvisi [13] (where the concept of regency is introduced).
The second strategy, Fast Paxos, tries to take advantage from a low throughput of the flow of initial values provided by the proposers. The approach presented by Lamport in [9] aims at reducing the number of communication steps to two, when the circumstances are favorable. If the previous consensus instance is finished for a long time and if all the active proposers[2] provide the same initial value, a gain can be obtained during the current consensus instance by just anticipating some part of the computation. To prepare the next consensus instance, a leader can send a special value, called an *Any* value, to the acceptors. Once an acceptor receives it, it is allowed to adopt a value directly provided by a proposer. As such an initial value does not pass in transit through the leader, the decision latency is reduced. As the values adopted by different acceptors during an attempt, are not necessarily equal, a more restrictive definition of quorums has to be used.

Within this paper, we provide a specification of the Multiple Integrated Consensus problem (called MIC for short). To solve efficiently the MIC problem, we present a protocol called Paxos-MIC that exhibits the following interesting features:
-1- It integrates, for the first time to our knowledge, within a single simple framework the two best known methods for reducing decision latency in Paxos-like protocols. The Paxos-MIC protocol always follows the strategy proposed in FastPaxos [1]. To also integrate the strategy of Fast Paxos [9], the leader asks the proposers to provide an initial value. If the proposers have no available value to send, the leader can decide at runtime (*i.e.*, in a dynamic manner) that an *Any* value will be used to reduce the latency.
-2- To control the flow of consensus instances, any coordinator has first to provide the decision value corresponding to the current consensus instance (via a call to the *DecidePush* function) to be able afterwards to obtain a new initial value for the next consensus instance (via a call to the *ProposePull* function). The calls are not initiated by the proposers.
-3- No reliable broadcast of a decision value is performed. In fact, a new consensus instance can start even if only one (possibly faulty) process is aware of the last decision value. The protocol ensures that this value will be logged by at least one correct process before the next decision is made. Indeed, while the protocol converges toward a new decision, it acts also to ensure the persistence of the previous decision without sending additional messages.
-4- We try to have a presentation of the Paxos-MIC protocol which remains very close from the terminology and the basic principles used in Paxos [8] and Fast Paxos [9]. Nevertheless, some choices that have conducted the design of our protocol, may lead the reader to have a better understanding and a slightly different look at all the contributions that we merge.
-5- We provide an analysis of the complexity of Paxos-like protocols, using as an evaluation metric, *the latency of reaching a decision*. Paxos-MIC performs the best in the described scenarios, by exploiting the optimizations used in both of the protocols it unifies. The practical interest of these scenarios is discussed by considering both the intrusion detection application [6] and the Grid application [11].

Road map: In Section 2 we present the system model and we provide a formal definition of the MIC problem. Related works are briefly discussed in Section 3. For the sake of clarity, the presentation of the Paxos-MIC protocol is done in two steps. In Section 4, we describe a first version of the protocol that does not take the existence of *Any* values into account. In Section 5, the modifications required to obtain the full version are presented. An analysis of Paxos-MIC is provided in Section 6, while Section 8 gives our conclusions.

2 The System Model and the MIC Problem:

System Model: We consider an asynchronous distributed system prone to crash failures. Different computing units are used to execute the n processes involved in the agreement protocol. At most f processes may crash. A correct process is a process that never crashes. We assume that there exists a majority of correct processes ($f<n/2$), which communicate by message passing. We assume bidirectional, fair lossy channels between every pair of processes. Processes operate at arbitrary speed and there exists no upper bound on message transfer delays. Messages can be duplicated and lost but not corrupted. To circumvent the FLP impossibility result [5], the system is extended with a leader election service that ensures the following property. Eventually a single correct process will be elected to be the leader, for sufficiently long time.

The Multiple Integrated Consensus Problem: The n processes involved in the Paxos-MIC protocol (coordinators and acceptors) interact with external proposers and external learners through two functions, named *ProposePull* and *DecidePush*. In both cases, the call is initiated by the Paxos-MIC protocol. During a consensus instance c, at least one and at most n coordinators will call (one or several times) the primitive *ProposePull*. This is done while they are acting as a leader. Each of

[2] In the best case, a single proposer is active.

them will receive either an initial value v (that may become a decision value $<v,c>$) or a special mark, $\perp$ or $\top$ (that cannot be selected to become a decision value). Any call to the *ProposePull* function leads at least one of the proposers to react. A reacting proposer has three possible choices. It can postpone its choice until the next call to the *ProposePull* function. In that case, it uses the special mark $\perp$ to invite the calling leader to call again this function later. Otherwise, the proposer can return immediately a definitive answer which is either an initial value or the special mark $\top$: the leader will never have to call again the *ProposePull* function during this consensus instance. When an initial value v is returned immediately, it becomes the initial value of the leader during the current consensus instance. Following the Paxos terminology, $\top$ is called an *Any* value. When the proposer returns $\top$, it makes a promise that at least one proposer will directly (*i.e.*, on its own initiative) provide an initial value v to the acceptors. Of course, a proposer makes this promise when no initial value is currently available. In that case, the acceptors adopt temporarily the special mark $\top$ until they can replace it with an initial value v. As shown by Lamport, a Paxos algorithm can be adapted to take advantage of this future delivery [9]. During consensus instance c, once the protocol has converged to a decision $<v,c>$, a call to the *DecidePush* function provides the new decision $<v,c>$ to the external learners. A process decides $<v,c>$ if it executes *DecidePush*$(<v,c>)$.

Formally, the Multiple Integrated Consensus problem is defined by four safety properties (three validity properties and an agreement property) and two liveness properties, that specify the process behavior (in fact, the leader's behavior in Paxos-MIC).
1. **Validity-Non-triviality**: If a process executes *DecidePush*$(<v,c>)$, then a process has previously executed *ProposePull*(c) and obtained either the initial value v or the special mark $\top$ replaced later by v.
2. **Validity-Atomicity**: If a process executes ProposePull(c) with $c>1$ then it has previously executed DecidePush$(<v,c'>)$ for all the value c' such that $1 \leq c' < c$.
3. **Validity-Unicity**: If a process executes several times ProposePull(c) then only the last call may return a value different from the special mark $\perp$.
4. **Uniform-Agreement**: If a process executes DecidePush$(<v_a,c>)$ while DecidePush$(<v_b,c>)$ is executed by another process, then $v_a=v_b$.
5. **Termination-Progress**: If a process that calls infinitely often ProposePull(c) can eventually obtain either an initial value v, or the special mark $\top$, then at least one process eventually executes DecidePush$(<v,c>)$
6. **Termination-Persistence**: If a process executes DecidePush$(<v,c>)$ then at least one correct process eventually executes DecidePush$(<v,c>)$.

The Uniform-Agreement property of the MIC problem is similar to that of any traditional consensus abstraction. This property specifies that two different values can not be decided for the same consensus instance.

The Multiple Integrated Consensus problem considers a sequence of consensus instances. Therefore, the classical Validity and the Termination properties need to be further extended to cope with the succession of consensus instances.

The Validity-Non-triviality property refers to a single consensus instance c and states that only an initial value v that has been provided by a proposer during the consensus instance c can be decided during that instance. The second Validity property ensures that two consecutive consensus instances do not overlap, from the upper layer application point of view. Indeed, any leader that tries to obtain a new value from a proposer for the current consensus instance, has first to become aware of the decision values corresponding to all the previous consensus instances. The Validity-Unicity property guarantees that, when a leader obtains a value different from $\perp$, it never calls again the *ProposePull* function during the current consensus instance. Thus, each leader can introduce at most one initial value during each consensus instance. The Termination-Progress property assumes that if a leader calls the *ProposePull* function infinitely often, it will receive an initial value. Under this assumption, the property states that at least one coordinator will decide. The second liveness property ensures that if a coordinator decides during a consensus instance c , then at least one correct coordinator will decide during c (maybe later). The name given to this property (Termination-Persistence) reflects the fact that, in the proposed solution, all the data needed to decide are kept by at least one correct acceptor and accessible to any (correct) coordinator. In the classical Consensus problem, any correct process eventually decides. Here, the two liveness properties just require that at least one of them decides. Obviously a stronger requirement can be satisfied by reliably broadcasting the decisions.

3 The Latency in FastPaxos and Fast Paxos

Overview of the Protocols: The Paxos protocol is based on a timestamp mechanism. When a coordinator is elected as a leader, it must determine the round number (also called ballot number) under which it will execute an attempt to converge towards the decision value. In Paxos-like protocols, a leader can interact with all the acceptors by broadcasting either a *Read* request (that contains only a round number) or a *Write* request (that contains both a round number and an attached value). In the Paxos terminology, a *Read* request is sent during a *Prepare* phase, while a *Write* request is sent during a *Propose* phase. As each acceptor keeps track of the highest round number ever observed, the round number contained in any request sent by a coordinator is used by an acceptor to discard old requests. A positive reply returned by an acceptor

contains the last value adopted by this acceptor as well as the round number during which this last update has been done. If a leader gathers enough positive replies during a *Prepare* phase r, it switches to the *Propose* phase r. But first it will use the information gathered during the *Prepare* phase to determine if it can adopt a value provided by a proposer or if it must select the most recent value among those provided by the acceptors. In the last case, the selected value has already been proposed by another coordinator acting also as a leader during the same consensus instance. Its selection is mandatory to ensure safety. Indeed, all Paxos-like protocols rely on the concept of Majority Quorum. To decide a value v_1, a learner must detect that a majority of acceptors have sent a positive reply in response to a *Write* operation sent by the same leader during the same *Propose* phase (the *Write* request contains a round number r_1 and a value v_1). To end a *Prepare* phase with a round number r_3 such that $r_3 > r_1$, a leader must gather a majority of positive replies. As two majority quorums intersect, among the positive replies to the *Read* operation, at least one acceptor indicates that its current value v_2 has been obtained during round r_2 with $r_1 \leq r_2 < r_3$. A reasoning by induction on the value r_3 allows to conclude that $v_1 = v_2$.

Latency: The *latency for reaching a decision* is defined by the number of communication steps that are performed between the two following events: "a value is available at a proposer" and "a decision value is acquired by a learner". In order to study the latency of the different protocols, we assume that a coordinator acts also as a learner and we consider two consecutive consensus instances, numbered $c-1$ and c. The latency is analyzed during consensus instance c, in three different scenarios. In the first scenario, the leader election service makes many mistakes and the leader often changes (asynchronous period). In the second and third scenarios, we consider an infinite stable period, in which the leader never changes (a unique coordinator can act as a leader during all the consensus instances). We also assume that no collision occurs: two proposers will not provide different direct values. In the second scenario, a value is provided by a proposer, during consensus instance c, as soon as the decision value corresponding to the consensus instance $c-1$ is learnt. In the third scenario, we consider the time interval between the end of the last consensus instance and the supply of an initial value by a proposer. We assume that this time interval is long enough to ensure that all the computation steps that can be executed in advance, are completed when the initial value is available.

In the first scenario (worst case), the maximal number of communication steps required by each protocol is finite, but not bound. In the second scenario, Classic Paxos [8] requires four communication steps. While the message sent by a proposer to the leader is in transit, the leader can perform in parallel, the first step of the *Prepare* phase. The replies of the acceptors for the *Prepare* phase and the two steps required by the *Propose* phase add three more message delays. In FastPaxos [1], the stable leader can perform directly the *Propose* phase, without having to execute the *Prepare* phase. The latency of reaching a decision is reduced to three communication steps: the proposal message sent by a proposer to the leader and the two communication steps required by the *Propose* phase. In the case of Fast Paxos [9], the latency remains equal to four, as both *Prepare* and *Propose* phases have to be executed before deciding.

Let us now consider the third scenario. In Classic Paxos [8], as the value provided by a proposer becomes available after the *Prepare* phase is completed, the latency is reduced to three communication steps: one corresponding to the message sent by the proposer and two other message delays required by the *Propose* phase. In FastPaxos [1], the cost remains equal to three, as the *Propose* phase cannot be executed in advance. In Fast Paxos [9], between the two consensus instances, the leader sends a special request, called an *Any* message, to inform the acceptors they are allowed to adopt a value provided directly by a proposer. When this value reaches the acceptors, the leader has already performed both the *Prepare* phase and sent the *Any* request. The latency decreases to only two communication steps: the message sent by a proposer to the acceptors and the notification messages sent by the acceptors to the learners. When the requirements of the third scenario are not satisfied, the expected benefit is not realized and, in some cases, a performance degradation can be observed [9].

4 The Paxos-MIC Protocol - Without *Any* values

Within this section, we provide a general description of the protocol without considering the use of *Any* values: a call to the *ProposePull* function returns either an initial value or the special mark $\perp$ (but never the *Any* value $\top$). Under this additional assumption, the proposed protocol satisfies the specification of the MIC problem and implements, in a simple way the first optimization proposed in FastPaxos [1]. Within the next section, we extend this protocol to cope with *Any* values. This two-step description allows to clearly identify which parts of the protocol are impacted by the second optimization proposed in FastPaxos [9]. Rather than describing directly the pseudo-code, we discuss first some key elements of the protocol.

Roles: The MIC protocol describes the behavior of only two types of entities: acceptor (denoted A_i) and coordinator (denoted C_i). A coordinator interacts with external proposers and external learners by calling respectively, the *ProposePull* and *DecidePush* functions, described in Section 2. In the proposed solution, a coordinator is also in charge of observing the decisions. Therefore, when it receives a feedback from an acceptor, part of its activity corresponds to a learning process.

The protocol consists of four Tasks. An acceptor executes two tasks called Task A and Task B (pseudo-code described in Figure 2), while a coordinator executes Task C (described in Figure 3) and Task D (described in Figure 4). Task A and C

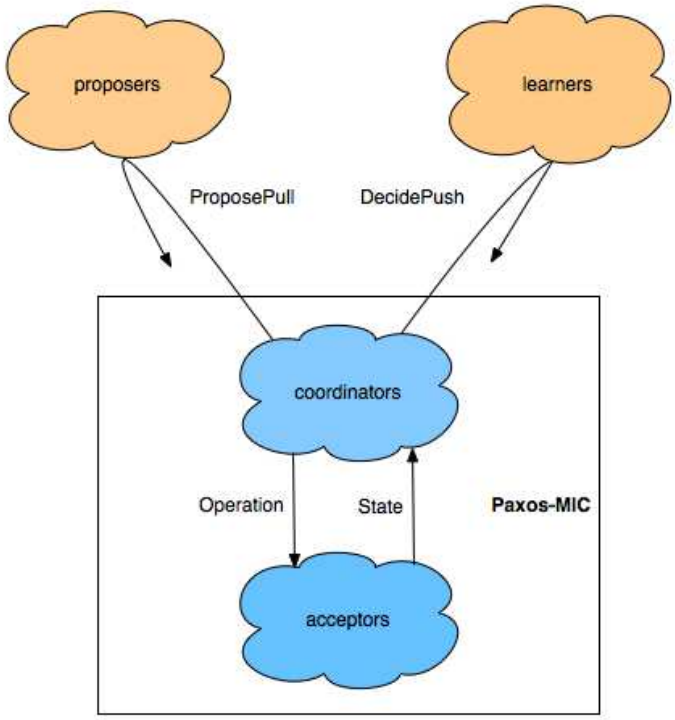

Figure 1: Communication Pattern

are executed upon the receipt of a message. Task B and D are executed periodically. All the statements contained in Task D can only be executed by a coordinator when it acts as a leader: for this reason, Task D is also called the Leader Task. In the next section, Figure 5 includes additional code executed by the acceptors and the coordinators to take *Any* values into account.

Messages: Coordinators and acceptors use two types of messages to interact: *Operation* and *State* messages. In the Figures, the shorter names *Op* and *St* are used.

A coordinator broadcasts its *Operation* messages to all the acceptors when it acts as a leader (Task D). An *Operation* message corresponds either to a *Read* operation (See line 18 of Task D), or to a *Write* operation (See line 16 of Task D).

An acceptor sends its *State* messages to the current leader (Task A, line 6 and Task B, line 8) and also to the initiator of the operation (Task A, line 6). A *State* message informs the recipients of the current state of the acceptor and it is also intended as a (positive or negative) acknowledgment by the initiator of an operation.

Note that the proposed interaction scheme (shown in Figure 1) is slightly different from the classical one adopted in Paxos-like protocols. In Paxos-MIC, *Read* and *Write* operations are not called "requests" because the sender does not wait for a reply. Even if an acceptor always sends a *State* message in reply to each *Operation* message it receives, such a query-reply communication pattern does not correspond to the well-formed requests used in Paxos and Paxos-like protocols. Indeed, we assume neither that a coordinator initiates only one query-reply at a time, nor that it waits for appropriate replies before proceeding to the next step. To cope with message losses, the last message sent is retransmitted as it reflects the last reached state of the sender. For this reason (among others), Tasks B and D are executed periodically. This choice simplifies the presentation and the analysis of the protocol. Of course, more efficient retransmission mechanisms can be used.

Variables: Table 1 provides a brief overview of the variables used in the code. This table indicates, for each variable, the role played by the process that manages it (acceptor A_i or coordinator C_j), a small description of its meaning and its assigned initial value. The list of variables is divided into three parts, based on the purpose they have in the protocol. The first part of the table presents the variables related to the use of *the leader election* mechanism, while the second part of the list comprises the variables involved in the management of *tags* and *tagged values*. Finally, the last part of the table describes the variables used for the *deciding*, *learning* and *logging* processes. Note that three variables ($SetRnd$, $SetCTag$ and $SetLid$) are used by a coordinator to manage sets of acceptors identities. In the code, if X is one of the three sets, any call to the function *Reset(X,false)* empties the set.

Table 1: Table of variables

Role	Name	Significance	Initial value
A_i	Lid	identity of the supported leader	1
C_j	$SetLid$	set of A_i that support $C_j$as leader	$\begin{cases} [true, .., true] & \text{if } j = 1 \\ [false, .., false] & \text{if } j \neq 1 \end{cases}$
	$RndLid$	leader's round number	j
A_i	Rnd	highest round ever observed	1
	$VTag$	highest tag ever observed	(0,0,0)
	$VVal$	value associated to $VTag$	$\perp$
C_j	Rnd	highest round ever observed	1
	$SetRnd$	set of A_i that reached Rnd	[*true*,..,*true*]
	$CTag$	highest tag ever observed	(1,1,0)
	$CVal$	set of the most recent tagged values	[$\perp$,...,$\perp$]
	$SetCTag$	set of A_i that sent CTag	[*false*,..,*false*]
A_i	$LogDVal$	array of logged values	[$\perp$,...,$\perp$]
C_j	$DVal$	last known decision value	$\perp$
	$PVal$	value used for a *Write* operation	$\perp$
	$PreparePhase$	current phase	*false*
	$LVal$	last recorded tagged value	$\perp$

Leader Election: A coordinator determines if it should act as a leader by relying indirectly on the information supplied by *a leader election service*. This service can be provided by a failure detector oracle Ω. The use of such a service ensures that a new process is selected as a leader when the current leader is suspected to have crashed. The protocol is indulgent: it never violates its safety properties even if, at the same time, multiple coordinators consider themselves leaders. The leader election service is invoked only by the acceptors and never by a coordinator. Task B of the protocol is periodically executed by an acceptor to query this service by executing a call to the function *GetLeader* (line 7). After learning the identity of its leader, an acceptor sends a *State* message to the chosen coordinator, thus announcing its support (line 8). Indeed, the first field $St.Lid$ of such a message contains the identity of a coordinator C_l chosen by the acceptor to be the current leader. A coordinator receives these messages and thus it is aware of the outcomes provided by up to n different leader election modules and passed in transit through different acceptors. A coordinator C_i does not store the identity of the leader selected by an acceptor A_j but just the fact that it has been chosen or not by this acceptor. Every time C_i receives a *State* message from an acceptor A_j, $SetLid[j]$ is set to *true* only if A_j considers C_i to be the leader (see Task C, line 1). C_i can act as the current leader only when it has gathered the support of a majority quorum of acceptors (see Task D, line 1). At the end of the initialization phase, the coordinator C_1 is supported by all the acceptors and thus it may act as the initial leader. A coordinator that has never been a leader remains quiescent as long as it does not obtain the support of enough acceptors. If a previous leader is no more supported by enough acceptors, it may continue to send *Operation* messages during a while. Yet, each time an acceptor receives one of them, a *State* message is returned (Task A). Consequently, a deposed leader will eventually discover that it should no more act as a leader.

In order to ensure progress, the leader election service must guarantee that, for all the acceptors, any call to the *GetLeader* function eventually designates the same correct coordinator. As a *State* message is periodically sent to the leader, a unique leader eventually obtains this stable and up-to-date information from a majority of acceptors, once all the old messages have been either lost or consumed.

Consensus Instances and Round Periods: A coordinator proceeds in a sequence of consensus instances and also in a sequence of round periods. Each consensus instance (respectively, each round period) in which a coordinator is involved, is identified by a consensus number (respectively, a round number). The division into consensus instances follows from the specification of the problem itself. The division into round periods results from the use of a leader election mechanism: a

```
Task A: When A_i receives msg Op(Rnd, Tag, Val, DVal) from C_j

      % Maintaining the most recent information ever observed
(1)   if ((Op.Tag.Rnd ≥ Rnd) ∧ (VTag ≺ Op.Tag) ∧ (Op.Val ≠ ⊥)) then
(2)       VTag ← Op.Tag;   VVal ← Op.Val;
(3)   endif;
(4)   if (Op.Rnd > Rnd) then Rnd ← Op.Rnd endif;

      % Logging decision values
(5)   LogDVal[Op.Tag.Con - 1] ← Op.DVal;
(6)   send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con - 1]) to C_j and C_Lid;

Task B: Periodically
      % Query the Leader Election Service
(7)   Lid ← GetLeader();
(8)   send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con - 1]) to C_Lid;
```

Figure 2: Protocol executed by an acceptor A_i

leader starts a new round period when it discovers that another coordinator has concurrently acted as a leader by executing a round period with a higher round number. Consensus instances and round periods are not linked : a consensus instance may span over several round periods and, conversely, during a single round period, several consensus instances may be solved. Two main counters, the consensus number (variable $CTag.Con$) and the round number (variable Rnd), mark the progress of a coordinator. These counters evolve independently. The current consensus number (variable $CTag.Con$) increases only when a new decision is made during Task C (line 6, 10, or 30). Indeed the participation of a coordinator C_i to a new consensus instance numbered c, can only start when the outcome of the previous instance, numbered $c-1$, is known by C_i. Note that the consensus number may decrease when line 15 of Task C is executed. The current round number (variable Rnd) is increased monotonically either during Task C (line 21) when a coordinator observes a higher round or during Task D (line 3), when a leader starts a new round period. At any time, the round number of a coordinator (variable Rnd) must be greater or equal to the highest round ever observed (variable $CTag.Rnd$). To fulfill this requirement, when it receives a message, a coordinator may adopt a round number already reached by another coordinator (Task C, line 21). When a coordinator C_i is elected as a new leader, it has to adopt the round number (also called ballot number by Lamport) under which it will execute attempts to converge towards decision values (Task D, line 3). This round number has to be strictly higher than the highest round ever observed by C_i. Moreover, another leader should not be able to use the same round number. In the proposed protocol, the variable $RndLid$ is used by C_i, to identify this particular round number. By construction (Task C, line 22), the value of $RndLid$ is obtained by adding the value i to a multiple of n. In this way, a round period numbered r is associated to a unique leader whose identity is equal to r `mod` n. While it acts as a leader, the two variables Rnd and $RndLid$ managed by a coordinator are equal (Task D, lines 2 and 3). When the coordinator is not acting as a leader, the variable $RndLid$ may be strictly greater than the variable Rnd.

Prepare and Propose Phases: A round period consists of an initial *Prepare* phase followed by a *Propose* phase. This distinction is only relevant when a coordinator acts as a leader (*i.e.* during an execution of Task D). Otherwise (*i.e.* during an execution of Task C), a coordinator takes its current round period number into account but it ignores the subdivision into two phases. A boolean variable $PreparePhase$ is used by a leader to know its current phase. This variable is set to true at the beginning of any round period and remains true till the *Propose* phase can start. After the initialization, the coordinator C_1 can execute immediately the *Propose* phase of the round period number 1. Apart from this particular case which is an optimization, any leader executing the *Propose* phase of a round has previously executed the corresponding *Prepare* phase. During the *Prepare* phase (respectively, *Propose* phase), a leader can generate *Read* (respectively, *Write* operations), while it executes Task D. The names given to the phases and the operations are similar to those used in the Paxos terminology introduced by Lamport [8].

The main purpose of a *Prepare* phase is to ensure that all the future *Write* operations of a leader will be consistent with those already performed by previous leaders. During this phase, a leader communicates its new round number to the acceptors (*Read* operations). If it gathers enough feedbacks from the acceptors, it can switch to the *Propose* phase (Task D, line 8). When this phase transition occurs, the leader uses the information collected during the *Prepare* phase to determine the origin of the value it will use during the first *Write* operation of its *Propose* phase (variable $PVal$). Only two cases are envisioned. If at least one acceptor has informed the leader of a value previously proposed by another coordinator acting also as leader during the same consensus instance, the leader must select one of the most recent values among such values (Task D, lines 7-9). In the basic version of the protocol, as we assume that no *Any* value is proposed, all these tagged values are necessarily equal and different from $\perp$ and $\top$. Otherwise, at the end of the *Prepare* phase, the value of $PVal$ remains equal to $\perp$. In that case, as any value can be chosen without posing a risk to violate the Uniform-Agreement property, the

```
Task C: When C_i receives msg St(Lid, Rnd, Tag, Val, DVal) from A_j
(1)  SetLid[j] ← (St.Lid = i);
     % Retrieving decision values of previous consensus instances
(2)  if (St.Tag.Con > CTag.Con) then
(3)     while (St.Tag.Con - 2 ≥ CTag.Con ) do
(4)           Dval ← RetrieveDec(CTag.Con);
(5)           DecidePush(< DVal , CTag.Con >);
(6)           CTag.Con ← CTag.Con +1;
(7)     enddo;
(8)     DVal ← St.DVal;  PVal ← ⊥;  LVal ← ⊥;
(9)     DecidePush(< DVal , St.Tag.Con - 1 >);  Reset(SetCTag, false);
(10)    CTag.Con ← St.Tag.Con;  CTag.Any ← 0;
(11) endif;
     % Maintaining the most recent information ever observed
(12) if (CTag ⪯ St.Tag) then
(13)    if (CTag ≺ St.Tag) then
(14)       if (St.Tag.Con < CTag.Con) then DVal ← St.DVal; endif;
(15)       CTag ← St.Tag;  Reset(SetCTag, false);
(16)    endif;
(17)    CVal[j] ← St.Val;  SetCTag[j] ← true;  LVal ← St.Val;
(18) endif;
(19) if (St.Rnd ≥ Rnd) then
(20)    if (St.Rnd > Rnd) then
(21)       Reset(SetRnd, false);  Rnd ← St.Rnd;
(22)       while (RndLid < Rnd) do RndLid ← RndLid + n enddo;
(23)    endif;
(24)    SetRnd[j] ← true;
(25) endif;
     % Deciding for the current consensus instance
(26) if (CTag.Any = 0) then
(27)    if ((Quorum_Maj(SetCTag)) ∧ (LVal ≠ ⊤)) then
(28)       DVal ← LVal;  PVal ← ⊥;  LVal ← ⊥;
(29)       DecidePush(< DVal , CTag.Con >);  Reset(SetCTag, false);
(30)       CTag.Con ← CTag.Con +1;  CTag.Any ← 0;
(31)    endif;
(32) else execute(code CAny);
(33) endif;
```

Figure 3: Protocol executed by a coordinator C_i

proposed value can be provided later by one of the proposers in response to a call to the *ProposePull* function (Task D, line 14).

During the following *Propose* phase, each *Write* operation executed by the leader aims to suggest a safe value to the acceptors. If enough acceptors follow this suggestion, this value becomes the decision value. Once the decision is learned (during the execution of Task C), the protocol goes on with the next consensus instance. While the leader remains stable, it will continue to execute the same round period and, more precisely, the same *Propose* phase. Let us notice that, once it enters the *Propose* phase, a leader broadcasts at most one *Write* message during each execution of Task D. Indeed, if it can propose neither a significant initial value nor an *Any* value $\top$, a leader sends no message. Otherwise, it periodically sends its last *Write* message until it makes a decision during Task C. Each time it decides, the proposed value is reset to $\perp$, the consensus number is increased and all the tagged values are discarded.

Tags and Tagged Values: A *tag* is defined as a triplet of integers denoted (r, c, a): the first integer r is a **R**ound number, the second integer c is a **C**onsensus instance number and the last integer a is only useful when **A***ny* values are used. At this stage of the explanation, we just need to know that this last integer is effectively a boolean variable, set to 0 or 1. When no *Any* values are used, it is always equal to 0. A *tagged value* v is defined as the result of a deliberate decision to associate a value v with a tag. We use the notation $(v, (r, c, a))$ to precisely refer to a tagged value v associated to the tag (r, c, a). In Paxos-MIC, we assume that a tagged value can be created (or declared) only at some well-defined stages of the computation, namely during the initialization phase and each time a new *Write* operation is executed during Task D. Indeed, only the leader is allowed to declare some new tagged values during the computation. Once it has been declared, a tagged value is propagated within the set of processes. A tagged value is contained in the *Operation* messages broadcast by a coordinator to all the acceptors and in the *State* messages sent by an acceptor to some coordinators. When a message is received, a copy of the transmitted tagged value can be stored temporarily in a set of four related variables.

With regard to a given consensus instance, an acceptor can store a single tagged value while a coordinator can store up to n tagged values that share the same tag. More precisely, an acceptor stores a tagged value in its variables ($VVal$,($VTag.Rnd$,$VTag.Con$,V This set of variables is initialized to $(\perp, (0, 0, 0))$. A coordinator can store a tagged value received from an acceptor A_j, in its set of variables ($CVal[j]$,($CTag.Rnd$,$CTag.Con$,$CTag.Any$)). This set of variables stores a tagged value if and only if the variable $SetCTag[j]$ is equal to true. When no *Any* values are used, all the values logged in $CVal$ by a coordinator are equal. Consequently, this common value is also contained in the variable $LVal$, which is used to keep the last recorded tagged value.

```
Task D: Periodically
      % If C_i can act as a leader
(1)   if (Quorum_Maj(SetLid)) then
          % If C_i has to start a new round period
(2)       if (Rnd < RndLid) then
              % Begin a Prepare phase
(3)           PreparePhase ← true;  Reset(SetRnd, false);  Rnd ← RndLid;
(4)       endif;
(5)       if (CTag.Any = 0) then
              % If C_i gathered enough feedbacks from A_j
(6)           if ((Quorum_Maj(SetRnd)) ∧ PreparePhase) then
                  % Select one of the most recent values
(7)               if (∃ k s.t. SetCTag[k]) then
(8)                   PVal ← CVal[k];  PreparePhase←false;
(9)               endif;
(10)          endif;
(11)      else execute(code DAny);
(12)      endif;
          % If C_i can execute the Propose phase
(13)      if (PreparePhase = false) then
              % If the value can be provided by a proposer
(14)          if (PVal = ⊥) then PVal ← ProposePull(CTag.Con) endif;
(15)          if (PVal ≠ ⊥) then
                  % Write operation
(16)              send Op(RndLid, (RndLid, CTag.Con, 0), PVal, DVal) to every A_k;
(17)          endif;
          % Read operation
(18)      else send Op(RndLid, CTag, LVal, DVal) to every A_k;
(19)      endif;
(20)  endif;
```

Figure 4: Protocol executed by a leader C_i

Lexicographical order: A lexicographical order is defined over the set of tagged values and denoted $\preceq$.
Let $(v,(r,c,a))$ and $(v',(r',c',a'))$ be two tagged values. Then, $(v,(r,c,a)) \preceq (v',(r',c',a'))$ if and only if $(r < r') \vee ((r = r') \wedge (c < c')) \vee ((r = r') \wedge (c = c') \wedge (a \leq a'))$. When $(v,(r,c,a)) \preceq (v',(r',c',a'))$ and $(r,c,a) \neq (r',c',a')$, the tagged value $(v',(r',c',a'))$ is said to be more recent than $(v,(r,c,a))$. In that case, we use the notation $(v,(r,c,a)) \prec (v',(r',c',a'))$. When a process receives a message which contains a tagged value, the tag of the received value and the tag of the value(s) already stored, are compared to determine the most recent one. Indeed, all the coordinators and all the acceptors adopt the same strategy: they only keep the most up-to-date tagged values, by taking into account only the messages that provide a more recent tagged value. Task A executed by an acceptor, manages the updating mechanism. An acceptor is a passive process which may only update its state when it receives an *Operation* message from a leader. It updates its tagged value (variables $VTag$ and $VVal$) at lines 1-3 and its round number (variable *Rnd*) at line 4.

A coordinator follows a similar behavior: during Task C, it updates its tagged values at lines 12-18 and its round number at lines 19-25. At any time, $SetCTag[j]$ is equal to true if and only if the coordinator has received from A_j the value contained in $CVal[j]$ associated with the tag $CTag$. Recall that, the variable $LVal$ is used to keep the value of the last recorded tagged value. When a coordinator observes an higher tag, it resets the variable $SetCTag$ before recording the value associated to the new tag. When a coordinator updates the value of the highest round number it has observed (variable *Rnd*), it also resets the list of acceptors that have reached this level (variable *SetRnd*). Note that the behavior of the acceptor is more restrictive because the test performed In Task A, at line 1 may lead to ignore a tagged value even if this tagged value is more recent than the logged one.

When they are not lost, messages are received in an arbitrary order by the coordinators and the acceptors. As the coordinators and the acceptors manage either monotonically increasing variables (*Rnd*, *RndLid*) or ordered tagged values ($CTag$, $VTag$), a receiver can easily determine which fields of the message provide a more recent information. Note that while the first field of a tag (the round number) is a monotonically increasing variable, the second field (the consensus number) may vary non-monotonically due to the use of the lexicographic order relation $\preceq$.

The computation of *RndLid* previously explained, tries to ensure that the leader's tagged value will be the most recent one, in the system. In this way, its proposed value has a chance of being accepted by at least a majority of the acceptors.

Operations: Only a leader is allowed to initiate an *Operation* during Task D: a *Write* operation at line 16 or a *Read* operation at line 18. An *Operation* message includes four fields, namely a round number r, a tag (r_p,c,a), a value v_p, and a value v_d. The value v_d contained in the last field, is the last known decision value obtained during the previous consensus instance numbered $c-1$. The value r, which is contained in the variable *RndLid*, is the round number currently used by the leader. During a *Prepare* phase, this value is necessarily strictly higher than r_p. Therefore, a simple test can distinguish a *Read* operation from a *Write* one. In the former case, $r > r_p$, while in the latter case, $r = r_p$. Even if an acceptor is able to distinguish a *Read* operation from a *Write* one, Task A treats an *Operation* message without checking its type. Roughly speaking, each received message is considered as a *Write* operation (lines 1 - 3) and then as a *Read* operation (line 4).

In the case of a *Read* operation, a new leader provides its new round period number, r. If an acceptor adopts this round number, it can no more consider tagged values with a lower round number, even if the received tagged value is more recent than its current one. Note that a similar rule is implemented in all the Paxos-like protocols. During a *Prepare* phase, a leader may execute several *Read* operations (Task D, line 18) but they all refer to the same round period. During a *Read* operation, the leader also relays a tagged value previously observed. Indeed, the tag (r_p,c,a) is contained in the variable $CTag$. It is the highest tag ever observed by a leader in a message received from an acceptor. All the tagged values logged in $CVal$ and, in particular the value v_p which is stored in $LVal$, are associated to this tag. As the tagged value $(v_p,(r_p,c,a))$ has been previously observed by an acceptor, it has been proposed in a past *Write* operation by the leader of round r_p. The new leader of round r has observed it and forwards it again during its *Read* operation. Note that at the beginning of the computation, due to the initialization, a leader may provide a tagged value equal to $(\perp,(1,1,0))$: this tagged value will not be considered by the acceptors, during Task A, due to the test performed at line 1.

In the case of a *Write* operation, the tagged value $(v_p,(r_p,c,a))$ contained in the message is a value proposed by the sender itself to become the next decision value. The value v_p is contained in the variable $PVal$ and the associated tag is defined by the triplet (*RndLid*,*CTag.Con*,0). The *Prepare* phase executed at the beginning of a round period ensures the correctness of the first *Write* operation. As explained before, at the end of the *Prepare* phase, if necessary, $PVal$ has been updated to be consistent with any previous attempt to converge to a decision value, made by another leader (Task D, line 8). Once the leader has decided during round period r, the following consensus instances cannot be in conflict with a previous attempt made by another leader. The fact that C_i may participate within the same round period to several consensus instances without executing again *Prepare* phases corresponds to an optimization similar to the one proposed in [1].

Decisions: By construction, the field $DVal$ of either a *State* or an *Operation* message, related to consensus instance c, contains the last known decision value, obtained during the previous consensus instance, numbered $c-1$.

In Paxos-MIC, during Task C, a coordinator acts also as an internal learner. Let c' be the current consensus number (variable $CTag.Con$) when a coordinator C_i starts the execution of Task C, and let c be the consensus number contained in the received message (field $St.Tag.Con$).

If $c>c'$ then C_i can conclude that it is behind with deciding: $c-c'$ decisions have already been made by other coordinators. During the execution of lines 2 - 11, in Task C, C_i makes up lost time in two steps. If more than one decision is missing, C_i will retrieve $c-c'-1$ decision values that were logged by at least a majority of acceptors. For each of them, it will execute the *RetrieveDec* function (Task C, line 4) and then call *DecidePush*$(< v_y, y >)$, where y varies from c' up to $c-2$ and v_y is the value (different from $\perp$) of a variable $LogDVal[y]$ managed by an acceptor. Then, in a second step, the last decision corresponding to the consensus instance $c-1$ is made by C_i at line 9. In that case, it executes *DecidePush*$(< DVal, c-1 >)$ where $DVal$ is the last known decision value contained in the received message. Note that, if a coordinator C_i decides $< v, c >$ either at line 5 or at line 9, then at least one coordinator C_j, has previously decided $< v, c >$ at line 29.

When $c' = c$, the tag contained in the *State* message received from A_j (field $St.Tag$), is possibly equal to the highest tag ever observed by C_i (variable $CTag$). In that case, the coordinator C_i is allowed to decide if it observes that a majority quorum of acceptors (including A_j) have adopted tagged values during the same round and the same consensus instance (Task C, lines 27 - 31). In the basic version of the protocol, as no *Any* value is used, all these tagged values are necessarily equal and different from $\perp$ and $\top$.

Logs: Old decision values are logged by the acceptors. The purpose of the logging mechanism is to ensure that for each decision value, at least one correct acceptor will be aware of it. Each acceptor A_i maintains an array of logged values, $LogDVal$. An entry k of this array is initialized to $\perp$ and used to store the decision value for consensus number k. A_i may become aware of this value which is contained in the field $DVal$ of any *Operation* message related to consensus number $k+1$. To decide a value during consensus $k+1$, a coordinator must observe that this value has been adopted by a majority of acceptors. Consequently, the previous decision contained also in these *Operation* messages, has necessarily been observed and logged by a majority of acceptors. At least one of them is correct and can provide this logged value, if necessary. The *RetrieveDec* function is used by a learner to obtain an old decision value. The execution of this function queries a majority of acceptors.

In the proposed solution, each time an acceptor A_i sends a *State* message related to the consensus number c, it includes in the last field of its message, the logged value corresponding to the consensus number $c-1$. Due to this choice, *Operation* and *State* messages have a similar structure. Moreover, it speeds up the retrieving of the last missing decision value.

The logs can be used to ensure the termination property which states that, during each consensus instance, at least one correct process eventually decides a value. Of course, in an asynchronous system, these logs might store an unbounded number of values. If weaker termination properties are considered, it is possible to implement amnesic logs that store only a limited number of decision values [4].

A different mechanism for logging decisions is used in [1]. A distributed structure, called a *round based register*, is used to log decision values. Each time a new consensus instance is started, a new register is created. Any correct process must be able to retrieve the decision value for any completed consensus instance. For achieving this purpose, the register instances must remain active even after the corresponding consensus instance has finished. The logged information is available and can be retrieved at any time. Such a logging mechanism requires that (possibly) a high number of register instances are kept active.

Retrieving Decision Values:

The *RetrieveDec* function takes as parameter, a consensus number c and returns the decision value corresponding to consensus number c. This function can be invoked by external learners or by a coordinator executing line 4 in Task C. The retrieving mechanism fetches decision values from the acceptors logs. Let us assume that a coordinator C_i invokes the *RetrieveDec* function with c as parameter, in Task C at line 4. C_i periodically broadcasts a request to obtain the decision value for consensus number c. This request message is sent to at least a majority of acceptors. Each acceptor A_j store the decision value for consensus number c, in the entry c of its logs (variable $LogDVal[c]$), only if A_j has received this value in an *Operation* message sent by a leader. If the message has not reached A_j, $LogDVal[c]$ stores the $\perp$ value. Each time A_j receives a request to access the c entry of its logs, it will provide the corresponding value only if this value is different from $\perp$. Once C_i receives a message containing a value different from $\perp$, it has successfully retrieved decision number c.

5 How to cope efficiently with *Direct Any* values

We present now the modifications that have to be done to obtain a Paxos-MIC protocol able to manage *Any* values. During a call to the *ProposePull* function made by a leader, a proposer that has no initial value to provide may allow the leader to start immediately a new consensus instance c with no real initial value (by returning $\top$). Proposers make a commitment to provide later all the acceptors with at least one significant value. Such a value is sent directly by a proposer to an acceptor and is called a *direct* value. The name *Any* value is used to refer to either the special mark $\top$ or to a direct value. Detecting

that a value v is an *Any* value is obvious when v is equal to $\top$. When v is a direct value, nothing allows to distinguish it from an initial value. To avoid this, the third field of the tag, called *Any*, is used to indicate whether the value adopted by an acceptor, is a direct value or not. In the case of a direct value, this field is equal to 1.

If the leader is currently executing the *Propose* phase of a round period r, when it obtains a $\top$ value, it can immediately broadcast *Operation* messages that contain the tagged value $(\top, (r, c, 0))$. An acceptor may adopt the value $\top$ as if it were a real initial value. To allow an acceptor to receive later a direct value from a proposer, we define an extension of the protocol executed by an acceptor. In Figure 5, the additional code executed by an acceptor is denoted Task A-Any. An acceptor A_i executes Task A-Any only when it receives a message from a proposer. This message contains both a consensus number, c and a direct value. The value should be an initial value (different from $\perp$ and $\top$). If the current tagged value $(v', (r', c', a'))$ of the acceptor is such that v' is equal to $\top$, $c' = c$, and r' is still the highest round number ever observed by the acceptor, then this direct value is used to replace the $\top$ value and the third field of the tag ($VTag.Any$) is set to 1. The above test ensures that the last operation that modifies the state of an acceptor, was a *Write* operation that contained an *Any* value, $\top$. An acceptor can adopt a direct value only if the condition $Rnd = VTag.Rnd$ remains true, which implies that the acceptor did not agree to participate in a higher round yet. The external write, done by the proposer, must not succeed if another operation with a higher round number has been initiated.

```
Task A-Any: When A_i receives msg P(Con, Val) from proposer P_k
(1)  if ((P.Val ≠ ⊥) ∧ (P.Val ≠ ⊤)) then
(2)      if ((VVal = ⊤) ∧ (VTag.Con = P.Con) ∧ (Rnd = VTag.Rnd))
(3)          then VTag.Any ← 1;   VVal ← P.Val;
(4)          send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con - 1]) to C_Lid;
(5)      endif;
(6)  endif;

code CAny:
(7)  if (Quorum_Any(SetCTag)) then
(8)      if (CollisionSafe) then
(9)          DVal ← the most frequent value CVal[k] such that SetCTag[k];
(10)         PVal ← ⊥;
(11)         DecidePush(< DVal , CTag.Con >);   CTag.Any ← 0;
(12)         Reset(SetCTag, false);   CTag.Con ← CTag.Con +1;
(13)     else if (Rnd = RndLid) then RndLid ← RndLid + n endif;
(14)     endif;

code DAny:
(15) if ((Quorum_Any(SetRnd)) ∧ PreparePhase) then
(16)     PVal ← the most frequent value CVal[k] such that SetCTag[k];
(17)     PreparePhase←false;
(18) endif;
```

Figure 5: Extension to the protocol to cope with *Any* values

Different proposers may provide different direct values. These proposal messages may be received in different orders by different acceptors. In such cases, a collision may occur. Unfortunately, different direct values will share the same tag $(r, c, 1)$. As indicated by Lamport [9], majority quorums have to be replaced by larger quorums called herein *Any* quorums. As an *Any* quorum is larger, the maximal number of failures f that are tolerated has to be lower. Moreover, these larger quorums are more difficult to obtain as they require to collect more replies from the acceptors. Thus, we may avoid to use them when it is not necessary. In Paxos-MIC the use of an *Any* quorum is mandatory if and only if the highest tag ever observed by the coordinator is associated to a direct value. The majority quorums evaluation performed in Task C, at line 27 (during the learning activity) and in Task D, at line 6 (during the *Prepare* phase) have not to be used if the current tag is associated to direct values. Note that the majority quorum used in Task D, at line 1 is not concerned by this problem. To determine if a majority quorum or an *Any* quorum has to be used, a test is performed at runtime at line 26 of Task C and at line 5 of Task D. If the evaluated condition is not true, either the code CAny or DAny is executed.

In [9], Lamport defines quorums as sets of processes (acceptors). Each round has a set of quorums associated with it. Classic rounds use classic quorums while fast rounds rely on fast quorums. These sets of acceptors must satisfy some given properties called *Quorum Requirements*. These properties state that (1) any two quorums must have a nonempty intersection and (2) any quorum and any two fast quorums from the same round must also have a nonempty intersection. Based on these requirements, Lamport has defined in [9], the minimum number of acceptors that can constitute a quorum. Let us assume that N is the total number of acceptors that are in the system. The cardinality of any classic quorum,Q_c, and of any fast quorum, Q_a, can be computed as follows:

$|Q_c| \geq \lfloor N/2 \rfloor + 1, \qquad |Q_a| \geq \lceil 3N/4 \rceil.$

The same requirements for quorums cardinalities are also obtained in [15].

As the value $\top$ is managed like any other initial value, we have to ensure that no coordinator C_i decides $< \top, c >$. Two possible cases are envisioned, regarding the most recent tagged values, logged in $CVal$. The *Any* field of the highest tag ever observed (variable $CTag.Any$) indicates which of the two cases occurs.

Case 1: If variable $CTag.Any$ is equal to 0, the coordinator C_i has not yet observed a *direct* value, in the messages received from the acceptors. In this case, all the tagged values logged in $CVal$ are equal. If the *Write* message containing the $\top$ value, has been lost, all the values are equal to the last initial value which was written by the previous leader and which is also kept in the last recorded tagged value, $LVal$. In this case, the classical majority quorum is enough to select the most frequent value and then decide it (Task C, lines 26 - 31). If all the values logged in $CVal$ are equal to $\top$ (in particular, the last recorded value $LVal = \top$), the acceptors have received the *Write* message, have updated their $VVal$ values to the special mark $\top$ and furthermore, none of the acceptors has accepted a *direct* value, yet. In this case, C_i must not decide the $\top$ value in Task C, at line 29. This is ensured by the test performed at line 27.

Case 2: If variable $CTag.Any$ is equal to 1, $CVal$ contains (possibly different) *direct* values. In this case, C_i must use an *Any* quorum for selecting the most frequent value from the ones logged in $CVal$ (code CAny, line 7). If collisions are rare, the values logged in $CVal$ may all be equal to some *direct* value, in which case this value can be safely chosen. It could also be the case that one particular *direct* value is frequent enough that it can be safely chosen. If this is the case, the predicate *CollisionSafe* becomes true and C_i can safely decide the chosen value (code CAny, lines 8 - 12). To define what "frequent enough" means, let us refer to the rule for choosing a value, defined in [15]. Let v be a *direct* value logged in $CVal$ and let T be the number of appearances of v in $CVal$. Assuming that Q_a is the *Any* quorum used in code CAny at line 7, the value v is defined to be frequent enough if at least a majority of acceptors inside the quorum Q_a have sent v in their messages. In other words, v is frequent enough if $T \geq \lfloor |Q_a| /2 \rfloor + 1$.

If none of the logged values is frequent enough, possible deadlocks due to the existence of collisions are detected and removed by forcing the start of a new round period (code CAny, line 13). If C_i remains the current leader, it will be force to begin the execution of a new *Prepare* phase, when it tests the condition at line 2 of Task D.

Note that during the *Prepare* phase executed by a new leader, the selected value can be the value $\top$. When this occurs, the value of the variable $CTag.Any$ is equal to 0 and thus all the tagged values stored in the data structure $CVal$ by the new leader are equal to $\top$. As the $\top$ value is selected at line 8, in Task D, the new leader will execute a *Write* operation with an *Any* value $\top$ like another leader before but with a higher round number. Proposers are expected to provide again their direct values till the end of this consensus instance.

In case of competing proposals, no value can be safely chosen. The usual way to recover from such a collision is to begin a new round. A coordinator C_i that learns of a collision in round i must start a new round with a number $j > i$, more precisely, C_i must initiate a *Prepare* phase by sending a *Read* request. In [9], Lamport suggests to optimize the classical mechanism to recover from collisions. However, this optimization is only possible under stronger assumptions. If i is a fast round and C_i is coordinator of rounds i and $i+1$, the information last sent during round i can be used during round $i+1$. Based on this observation, C_i can skip the *Prepare* phase for round $i+1$ as it knows that no one else has acted as a leader between rounds i and $i+1$. Therefore, round $i+1$ can begin directly with the *Propose* phase. Two *collision recovery* mechanisms are described in [9], namely *coordinated* and *uncoordinated* recovery.

6 Analysis of Paxos-MIC

Let us consider again the three scenarios from Section 3. Like all the other protocols, in the first scenario, the Paxos-MIC protocol requires a finite, but not bound, number of communication steps. In the other two scenarios, Paxos-MIC performs the best, by exploiting the optimizations used in both of the protocols it unifies. It allows a latency of three message delays (the same as [1]), in the second scenario, and a latency of two communication steps (similar to [9]), in the third one. In Paxos-MIC, a leader obtains a new proposal by invoking the ProposePull function. Any call to this function sends a request message to the external proposers which determines at least one of them to react by sending a proposal message to the initiator of the request. However, the step required by the request message does not add an extra delay to the overall latency. Only the reply of the proposer is considered when the latency is computed. Indeed, if the proposers are not acting as learners, they do not require to know previous decision values in order to compute a new proposal. In this case, proposers may send a proposal message on their own initiative, without having to wait for a request from a leader. However, proposers may also act as learners and compute their proposals based on previous decision values. We can make the assumption that a call to a DecidePush function may also determine the proposers to send a proposal message to the leader. Thus, any call to the DecidePush function has also the effect of a call to a ProposePull function. Each time a decision value is provided to a learner, a proposal can be computed for the next consensus instance and sent in a reply message. Thus, the request to provide a value, sent to the proposers does not add an extra communication step to the latency.

In [6], we investigate the use of diversified web servers to detect intrusions corresponding to unknown attacks. Each http request is executed simultaneously on different web servers. By assumption, these servers exhibit different vulnerabilities and thus an attack (against integrity or confidentiality) may succeed on at most one of them. A consensus service is used at various levels of the proposed architecture, to create a total order on the incoming requests and to maintain consistency within different sets of replica. In [11], we consider a Grid that federates resources provided by different institutions. We propose consensus-based control mechanisms, first to cope efficiently with the dynamic changes of the computing capacity of the Grid (even if these changes are unpredictable, in the case of crash failures) and second, to distribute the tasks among the resources in an efficient way (dynamic load balancing).

In these two applications, most of the time, the system is rather synchronous and failures are rare. Scenario 1 is likely to happen. Obviously, both application benefit from the FastPaxos optimization. In the case of the intrusion detection application, we analyzed the logs corresponding to the request addressed to the web server of an engineering school, during one month. Periods of low activity are frequent and correspond to scenario 3. Consequently, this application may also benefit from the Fast Paxos optimization.

7 Proof

Lemma 1. *If a leader C_i executes, in the following order, a Propose phase numbered r_1, a Prepare phase numbered r_2, and again a Propose phase numbered r_3, then $r_1 < r_3$.*

Proof A round period begins with a *Prepare* phase, possibly followed by a *Propose* phase. The two phases of a same round period are identified by the same round number. By definition, during a round period r executed by a leader C_i, the values of the variables *Rnd* and *RndLid* managed by C_i are both equal to r. When the values of these two variables are different, no round period is currently executed by the leader. Obviously, the condition $Rnd \leq RndLid$ is an invariant. A new round period starts when a leader executes line 3 of Task D. In that case, the value of the variable *Rnd* (*i.e.*, the lowest one) is set to the value of the variable *RndLid* (*i.e.*, the highest one). A round period ends once the predicate $Rnd \neq RndLid$ becomes true again. This can only occur when an update of the variable *RndLid* is performed (either at line 22 of Task C or at line 13 of the code *CAny*). In both cases, the variable *RndLid* increases by a multiple of n. As the two variables *Rnd* and *RndLid* are monotonically increasing, we have $r_1 \leq r_2 \leq r_3$. If the *Propose* phase r_1 occurs before the execution of the *Prepare* phase r_2, these two phases cannot belong to the same round period. Thus $r_1 + n \leq r_2$ and consequently $r_1 < r_3$. $\square_{Lemma\ 1}$

Lemma 2. *Any message sent during the computation contains both a round number r and a tagged value $(v, (r', c, a))$ such that $r \geq r'$. Furthermore in the case of a Write Operation message, $r = r'$, while in the case of a Read Operation message, $r > r'$.*

Proof Two types of messages are sent: *Operation* messages and *State* messages. Each of them includes a round number r and a tagged value $(v, (r', c, a))$. Within this proof, a message is said to be correct if $r \geq r'$. Otherwise the message is incorrect. We have to demonstrate that all the messages exchanged during the computation are correct.

In the case of a *Write* operation, a leader C_i broadcasts its *Write Operation* message when it executes line 16 of Task D. We have necessarily $r = r'$. Actually, r and r' are both corresponding to the value of the variable *RndLid* managed by the coordinator C_i. So any *Write Operation* message is correct.

In the case of a *Read* operation, a leader C_i broadcasts its *Read Operation* message when it executes line 18 of Task D. At this moment, r is the current value of its variable *RndLid* and (r', c, a) is the current value of its variable *CTag*. The coordinator C_i is currently executing the *Prepare* phase of its round period r. If a *Prepare* phase is in progress, we have necessarily $r > 1$. Indeed the round period numbered 1 (if any) does not require the execution of a *Prepare* phase by its leader C_1. During a *Prepare* phase, the values of the variables *RndLid* and *Rnd* managed by C_i are necessarily equal (See line 3 of Task D). So r is also the value of the variable *Rnd* when the *Read Operation* message is broadcast by C_i. The value of the variable *CTag* managed by C_i can change only when line 15 of Task C is executed. If C_i has never updated this variable since the initialization phase, we have $(r', c, a) = (1, 1, 0)$. So, as $r > 1$, we have $r > r'$. Otherwise, the variable *CTag.Rnd* has been changed at least once during an execution of line 15 of Task C. Let us consider the sequence of all the executions of Task C that have been performed before the current execution of Task D. During some executions of Task C, C_i has changed the value of its variable *CTag.Rnd* (See lines 12-18) and during some executions of Task C (not necessarily the same ones), C_i has changed the value of its variable *Rnd* and also sometimes the value of its variable *RndLid* (See lines 19-25). We have to demonstrate that the property $Rnd \geq CTag.Rnd$ is satisfied at the end of each execution of a Task C. We have already shown that this property is true before the first execution of Task C. The variable *Rnd* can be updated either at line 21 during Task C or at line 3 during Task D. In both cases, the new value is strictly higher than the previous one (See the condition

evaluated just before the update occurs). Therefore, if the property mentioned above is true at the end of an execution of Task C, the property still holds when the next execution of Task C starts. Let us consider a particular execution of Task C. Let r_1 and r'_1 (respectively r_2 and r'_2) be the values of variables *Rnd* and *CTag.Rnd* before (and respectively after) the execution of this task. Let us assume that $r_1 \geq r'_1$. If line 15 is not executed, the property still holds at the end of Task C: $r_2 \geq r'_2$. Indeed, we have $r_2 \geq r_1$ and $r'_2 = r'_1$. But if the value of the variable *CTag.Rnd* is modified at line 15, it may be the case that $r_2 < r'_2$ even if $r_2 \geq r_1$ and $r'_2 \geq r'_1$. This scenario may occur if the received *State* message which contains the fields $St.Rnd \leq r_2$ and $St.Tag.Rnd = r'_2$ is incorrect: $St.Rnd < St.Tag.Rnd$. At this stage of the proof, we conclude that a coordinator may generate an incorrect *Read Operation* message only if it has received previously an incorrect *State* message from an acceptor. But, if we assume that all the received *State* messages are correct, the property $Rnd \geq CTag.Rnd$ is satisfied when C_i broadcasts the *Read Operation* message: $r \geq r'$. Moreover, the leader C_i is the only coordinator able to create a tagged value with a round number equal to r. This can only be done during the *Propose* phase of the round period r. Consequently while C_i is still executing the *Prepare* phase of the round period r, no acceptor may have already adopted a tagged value with a round number equal to r: at this time the adopted tagged value of any acceptor has a tag that includes either a strictly lower or a strictly higher round number. Consequently, in the case of a *Read Operation* message, $r \neq r'$ and so $r > r'$.

A similar demonstration can be conducted in the case of a *State* message sent by an acceptor A_j. A *State* message is sent either during Task A (at line 6), during Task B (at line 8) or during Task A-Any (at line 4). In the three cases, when a *State* message is sent, r is the current value of the variable *Rnd* managed by A_j while r' is the current value of its variable *VTag.Rnd*. During the initialization phase performed by A_i, the variable *Rnd* is set to 1 while the variable *VTag.Rnd* is set to 0. Consequently, if the acceptor A_i sends a *State* message without having executed before line 2 of Task A, we have necessarily $r > r'$. Otherwise, the variable *VTag.Rnd* has been changed at least once during an execution of line 2 of Task A. Let us consider the sequence of all the executions of Task A that have been performed before the sending of the *State* message. If the *Operation* message received during Task A is correct, we have $Op.Rnd \geq Op.Tag.Rnd$. Consequently, if the condition evaluated at line 1 is satisfied, the property $Op.Rnd \geq Rnd$ is true when the condition at line 4 is tested. Thus when the variable *VTag.Rnd* is updated with the value of *Op.Tag.Rnd*, the value of the variable *Rnd* at the end of the task is necessarily equal to *Op.Rnd* and so $Rnd \geq VTag.Rnd$. Consequently, if all the *Operation* messages received during previously executed Tasks A are correct, the *State* message sent by A_i is such that $r \geq r'$. But if one of these *Operation* messages is incorrect, we may have $r < r'$. We conclude that an acceptor can send an incorrect message only if it has received previously an incorrect message from a coordinator.

Obviously our previous intermediate conclusions show that the Lemma holds. Indeed, while it has received no message, a coordinator or an acceptor sends only correct messages. Otherwise, an incorrect message can be send only if another incorrect message has already been received by the sender. As nobody is able to send an initial incorrect message, all the messages are correct. $\square_{Lemma\ 2}$

Lemma 3. *Any message contains a tagged value* $(v, (r', c, a))$ *and satisfies one of the three following conditions.*

- $v = \perp$
  In this first case, the message is either a State message such that $(r', c, a) = (0, 0, 0)$ *or a Read Operation message such that* $(r', c, a) = (1, 1, 0)$.
- $v \neq \perp$ *and* $a = 0$
  In this second case, a coordinator has previously broadcast at least one Write Operation message that contains the tagged value $(v, (r', c, a))$.
- $v \neq \perp$ *and* $a = 1$
  In this third case, we have necessarily $v \neq \top$ *and a coordinator has previously broadcast a Write Operation message that contains the tagged value* $(\top, (r', c, 0))$.

Proof

Note that the three cases are distinct and cover all the possible situations.

Case 1: If $v = \perp$, the message is not a *Write Operation* message. Indeed before sending such a message, a leader checks if the value v of its variable *PVal* is different from $\perp$ (See line 15 of Task D).

An acceptor A_i can change its tagged value during the execution of either Task A or Task A-Any. In both cases, its new value cannot be equal to $\perp$ due to the test performed at the very first line of each of these tasks. Consequently, if a *State* message sent by A_i contains a value v equal to $\perp$, A_i has never changed its tagged value before. The message contains the initial value $\perp$ and the initial tag $(0, 0, 0)$.

In the case of a *Read Operation* message, a coordinator C_i broadcast the message during the execution of line 18 of Task D. At this moment, v is the value of the variable *LVal* while (r', c, a) is the current value of the variable *CTag*. Both

variables can only be updated during the execution of lines 12-18 of Task C. If C_i has never updated its variable *LVal* since the initialization phase, it has also never updated its variable *CTag*. In this particular case, $v = \perp$ and $(r', c, a) = (1, 1, 0)$. Furthermore, C_i cannot adopt the value $\perp$ during an execution of line 17 of Task C. As demonstrated before, if a received *State* message contains the value $\perp$, the associated tag is equal to $(0, 0, 0)$. Consequently, due to the fact that the variable *CTag* can only increase according to the order relation $\preceq$, the condition expressed at line 12 of Task C cannot be satisfied: $(0, 0, 0) \prec (1, 1, 0) \preceq CTag$. This leads us to validate the first part of the Lemma.

Case 2: If the message is a *Write Operation* message, the property holds trivially.

If the message is a *Read Operation* message, its broadcast is done by a coordinator C_i when it executes line 18 of Task D. Let us assume that the leader C_i is currently executing the *Prepare* phase of the round period r_1. v is the current value of its variable *LVal* and (r', c, a) is the current value of its variable *CTag*. As $v \neq \perp$, the tagged value of C_i is no more its initial one. Line 17 of Task C has been executed by C_i at least once. The leader C_i includes in its *Read Operation* message a tagged value $(v, (r', c, a))$ which is in fact a copy of a tagged value previously contained in a *State* message received by C_i from an acceptor A_j. As $a = 0$, the acceptor A_j has not updated its tagged value $(v, (r', c, a))$ during an execution of the Task A-Any. Moreover, as v is not equal to $\perp$, the tagged value $(v, (r', c, a))$ is not the initial tagged value of A_j. The acceptor A_j has adopted the tagged value $(v, (r', c, a))$ during an execution of Task A when it has received an *Operation* message from a coordinator C_k. At the end of this task, the round number r_2 contained in the variable *Rnd* managed by A_j was such that $r_1 > r_2 \geq r'$. Indeed, due to Lemma 2, we have $r_2 \geq r'$. Moreover, by construction, when C_i executes the *Prepare* phase of the round period r_1, its round number r_1 is strictly higher than the highest round number ever observed before (at least equal to r_2). If the *Operation* message broadcast by C_i is a *Write Operation* message, the property holds. Otherwise, we can iterate the reasoning. The leader C_k has broadcast a *Read Operation* message while it was executing the *Prepare* phase of the round period r_3. Again there exists an acceptor A_l that has adopted the tagged value $(v, (r', c, a))$ during an execution of Task A. Let us assume that, at the end of this Task A, the round number contained in the variable *Rnd* managed by A_l was equal to r_4. The following property has to be satisfied: $r_1 > r_2 \geq r_3 > r_4 \geq r'$. At each iteration, a coordinator can broadcast a copy of a copy of a tagged value previously stored by an acceptor. But as the number of iterations is limited by the fixed bound r', there exists at least one coordinator who has broadcast a *Write Operation* message which includes the tagged value $(v, (r', c, a))$ during a round period r such that $r_1 \geq r \geq r'$.

If the message is a *State* message, an acceptor A_i has sent it either during Task A (at line 6) or during Task B (at line 8) but not during Task A-Any (because a is still equal to 0). In the two cases, when a *State* message is sent, v is the current value of the variable *VVal* managed by the acceptor A_i while (r', c, a) is the current value of its variable *VTag*. As v is not equal to $\perp$, the tagged value $(v, (r', c, a))$ is not the initial tagged value of A_i. Thus, the tagged value has been adopted by A_i during an execution of Task A when the acceptor has received a *Read Operation* message that contains the tagged value $(v, (r', c, a))$. If such a message exists, we have previously demonstrated that there exists at least one coordinator who has broadcast a *Write Operation* message which includes the tagged value $(v, (r', c, a))$.

Case 3: If $a = 1$, the message is not a *Write Operation* message. Indeed when a leader sends such a message, the value of a is always equal to the constant value 0 (See line 16 of Task D). Thus, in case 3, the message is either a *Read Operation* message or a *State* message. In a first step, we demonstrate the following property: if a *Read Operation* message or a *State* message contains the tagged value $(v, (r', c, a))$ with $a = 1$ then necessarily at least one acceptor A_k has previously executed Task A-Any and its tagged value was equal to $(v, (r', c, a))$ after the execution of the line 3 of this task .

We will prove this property by contradiction. Let us assume that some messages contain the tagged value $(v, (r', c, a))$ but that no acceptor has previously created this tagged value during an execution of the Task A-Any. In the case of a *Read Operation* message, the message is broadcast by a leader C_i when it executes line 18 of Task D. As the value of its variable *LVal* is different from $\perp$, the coordinator C_i has previously executed at least once the Task C. The tagged value $(v, (r', c, a))$ included by C_i in its *Read Operation* message is in fact a copy of a tagged value previously contained in a *State* message received by C_i from an acceptor. In the case of a *State* message, the message is sent by an acceptor A_j either during Task A (at line 6) or during Task B (at line 8). The sending of the message cannot occur during Task A-Any (at line 4): otherwise this contradicts our assumption. As the value of the variable *VVal* managed by A_j is different from $\perp$, the acceptor A_j has previously executed at least once the Task A. The tagged value $(v, (r', c, a))$ included by A_j in its *State* message is a copy of a tagged value previously contained in a *Read Operation* message received by A_j from a coordinator (and not the result of an update done during an execution of the Task A-Any). If some coordinators and some acceptors include the tagged value $(v, (r', c, a))$ in their *Read Operation* messages and their *State* messages, the iterative reasoning conducted during the analysis of case 2 can again be applied. For similar reasons, we conclude again that copies of the tagged value can be exchanged only a finite number of times. But, as $a = 1$, no coordinator has initially broadcast a *Write Operation* message which includes the tagged value $(v, (r', c, a))$. So this contradict our assumptions. Either no message contains the tagged value $(v, (r', c, a))$ (and the lemma holds) or there is at least one acceptor A_k that has created the tagged value $(v, (r', c, a))$ during an execution of Task A-Any.

Let us consider the behavior of an acceptor A_k that updates it tagged value during Task A-Any. During an execution of this task, only the value itself and the third field of the associated tag can be modified (See line 3 of Task A-Any). If

$(v,(r',c,a))$ is the tagged value of A_k when it ends the execution of Task A-Any, we use the notation $(v',(r',c,a'))$ to refer to the tagged value of A_k at the beginning of this task. A_k can accept a direct value v during Task A-Any only if the value of v is different from $\top$ (See line 1 of Task A-Any) and if the previous value v' of its variable *VVal* is equal to $\top$ (See line 2 of Task A-Any). As this variable is initialized to $\perp$ and as a direct value equal to $\top$ cannot be accepted during an execution of Task A-Any (See line 1 of Task A-Any), the acceptor A_k has previously adopted the tagged value $(\top,(r',c,a'))$ during an execution of Task A. When an update occurs during an execution of Task A, all the fields of the tagged value managed by A_k are simultaneously updated (See line 2 of Task A). Therefore, during this execution of Task A, $(\top,(r',c,a'))$ denotes also the tagged value contained in the *Operation* message received from a coordinator. If $a'=1$, we face again the same contradiction: on one hand, the tagged value $(\top,(r',c,1))$ cannot be contained in a *Write Operation* message and, on the other end, an execution of the Task A-Any cannot create this tagged value. The above discussion leads us to conclude that a tagged value $(\top,(r',c,1))$ can never exist. Thus, $a'=0$ is the only possible scenario. The demonstration already conducted in case 2 ensures that at least one *Write Operation* message with the tagged value $(\top,(r',c,0))$ has been broadcast if an *Operation* message with the same tagged value exists. The Lemma holds.

$\square_{Lemma\ 3}$

Lemma 4. *When an acceptor updates its tagged value, its new value is always more recent than the previous one.*

Proof The tagged value of an acceptor A_i is defined by two variables: (*VVal*,(*VTag*)). An acceptor can update its tagged value either during the execution of Task A when it receives a message from a coordinator or during the execution of Task A-Any when it receives a message from a proposer. Let $(v,(r,c,a))$ be the tagged value of an acceptor A_i when it starts to execute one of these two tasks and let $(v',(r',c',a'))$ be its tagged value at the end of this task. We have to demonstrate that:
$(v,(r,c,a)) \neq (v',(r',c',a')) \Rightarrow (r,c,a) \prec (r',c',a')$

First we consider that the acceptor A_i has updated its tagged value during an execution of line 2 of Task A. If an update is performed during Task A, $(v',(r',c',a'))$ denotes also the tagged value contained in the *Operation* message received from a coordinator. Line 2 of Task A is executed only if the condition expressed at line 1 is satisfied. In that case, we have $(r,c,a) \prec (r',c',a')$ and consequently the received tagged value is more recent than the tagged value previously stored by A_i.

Let us now consider that the acceptor A_i has updated its tagged value during an execution of the Task A-Any. If the tagged value of A_i is updated, we have $v=\top$ (See line 2 of Task A-Any) and $a'=1$ (See line 3). The other fields of the tag remain unchanged: $r=r'$ and $c=c'$. The tagged value $(\top,(r,c,a)$ has been adopted by A_i during an execution of Task A. As this tagged value was contained in an *Operation* message, due to Lemma 3, we can conclude that $a=0$. The fact that $0=a<a'=1$ leads us to conclude that $(r,c,a) \prec (r',c',a')$.

$\square_{Lemma\ 4}$

Remarks related to Lemma 4:

- During the execution of Task A, an acceptor that receives a tagged value $(v',(r',c',a'))$ may keep its previous tagged value even if the received one is more recent. This occurs if the value of r' is strictly less than the current round number of the acceptor.
- If we consider the second field of a tag which is corresponding to the consensus number, an acceptor whose previous tagged value was equal to $(v,(r,c,a))$ can regress by accepting a received tagged value $(v',(r',c',a'))$ such that $r'>r$ but $c'<c$. More precisely, an acceptor can adopt a value related to consensus c and later accept a value related to consensus $c-1$. Nevertheless, the new value is more recent than the previous one according to the order relation $\preceq$.

Lemma 5. *After it resets its log of values, a coordinator can only adopt more recent tagged values. Moreover the first field of its tag, namely the round number logged in the variable CTag.Rnd can never decrease while the second field of its tag, namely the consensus number logged in the variable CTag.Con, can only decrease if the first field increases.*

Proof Recall that a coordinator C_i stores a single tag (in its variable *CTag*) and up to n tagged values (in its variable *CVal*). If k is such that $1 \leq k \leq n$, the value contained in the entry *CVal*[k] is valid if and only if the corresponding boolean entry *SetCVal*[k] is equal to *true*. The variables *CTag*, *CVal*, and *SetCVal* can only be modified during the execution of Task C.

All the logged values are removed when the statement "*Reset(SetCTag,false)*" is executed. This occurs during the execution of either line 9 of Task C, line 15 of Task C, line 30 of Task C or line 12 of the code CAny called at line 32 of Task C. During a same instance of Task C, a coordinator can execute both line 9 and line 15. But if it executes one of these lines, it can execute neither line 30 nor line 32. Indeed, if the list *SetCTag* is emptied either at line 9 or at line 15, a majority quorum cannot be observed immediately after: at line 27 in Task C and at line 7 in Task C-Any. Moreover, line 30 and line 32 are mutually exclusive instructions (See the If-Then-Else statement at lines 26-32).

Let (r, c, a) be the value of the variable *CTag* when C_i starts the execution of Task C (*i.e.*, r is the value of the variable *CTag.Rnd*, c is the value of the variable *CTag.Con* and a is the value of the variable *CTag.Any*). Let (r', c', a') be the value of the tag *St.Tag* contained in the *State* message received by C_i. Finally, let $(r", c", a")$ be the value of the tag *CTag* of C_i at the end of Task C. When C_i resets its log to the empty set, we have to prove that $(r, c, a) \prec (r", c", a")$. If this property is satisfied, any tagged value stored later is more recent: each modification of the tag contained in the variable *CTag* implies a reset of the log. Note that a later storage can occur either during the same execution of Task C or during any future execution of Task C. Indeed the log of a new tagged value is done either during the execution of line 10 of Task C or during the execution of line 17 of Task C. So, a more recent value is immediately stored if the log is reset to the empty set either at line 9 or at line 15. Otherwise, if the reset is done during the execution of line 30 or line 32, the log is still empty when Task C ends.

C_i executes line 9 only if the condition $(c' > c)$ holds. In that case, only the second field of the tag is modified: $c" = c'$. If the tag is not modified again at line 15, we have $((r" = r) \wedge (c" > c))$. Thus, in that case, $(r, c, a) \prec (r", c", a")$. Now if the whole tag is modified at line 15, we have $(r", c", a") = (r', c', a')$. The condition $(r, c, a) \prec (r', c', a')$ has been evaluated to *true* just before, at line 13. Consequently, $(r, c, a) \prec (r", c", a")$. When C_i executes either line 30 of Task C or line 12 of the code CAny, the first field of the tag remains unchanged: $r" = r$; the second field is increased by 1: $c" = c + 1$. Consequently, $(r, c, a) \prec (r", c", a")$.

In all cases, $(r" \geq r)$. Thus the first field of the tag never decreases. Now, let us assume that the second field of the tag decreases. This cannot occur when line 30 or line 32 is executed: in both cases, the consensus number is increased by 1. As mentioned before, if C_i executes line 9, we have $c" = c'$ and, due to the test evaluated at line 2, $c' > c$. Consequently, in that case also, the second field also increases: $c" > c$. When C_i executes line 15 (without having executed line 9 just before), it could be the case that $c" < c$. Due to the test evaluated at line 13, we have $(r, c, a) \prec (r', c', a')$. So if $c' < c$, the condition $r < r'$ must be satisfied. We conclude that $r < r"$ when $c" < c$.

$\square_{Lemma\ 5}$

Lemma 6. *If c is the consensus number stored in the variable CTag.Con managed by a coordinator C_i, then either c is equal to 1 or C_i has already executed DecidePush $(< d", c" >)$ for all the consensus instance $c"$ such that $1 \leq c" < c$.*

Proof The variable *CTag.Con* is initialized to 1. Let us consider a particular execution of Task C during which the value of the consensus number of C_i is increased. Let us denote respectively by c' and c the values of the variable *CTag.Con* respectively at the beginning and at the end of this task. If $c' < c$, an update of the tag has been done either at line 9, at line 30, or at line 32 of Task C (See the line 12 of the code C-Any). Note that an update may also occur at line 15. But in that case, as $c' < c$, line 10 has been executed just before during the same task C. Let us now analyze the three possible cases.

If C_i executes line 9, it has also executed the line 8 just before and $c - c' - 1$ times the line 5 during the previous while loop. If C_i executes line 30, it increases its consensus number by 1: $c' = c - 1$. Moreover it has necessarily executed the statement *DecidePush*$(< d', c - 1 >)$ at line 29. A similar analysis is conducted when C_i executes line 12 of code C-Any. It increases also its consensus number by 1: $c' = c - 1$. Moreover it has necessarily executed the statement *DecidePush*$(< d', c - 1 >)$ at line 11 of the code C-Any.

As the same reasoning can be applied during any execution of Task C that increases the consensus number, the Lemma holds: If the round number of C_i is equal to c, C_i has executed the statement *DecidePush*$(< d", c" >)$ for all the value of $c"$ such that $1 \leq c" < c$.

$\square_{Lemma\ 6}$

Lemma 7. *A coordinator cannot broadcast two Write Operation messages that contain the same tag but two different tagged values.*

Proof A leader broadcasts a *Write Operation* message when it executes line 16 of Task D. Consider a coordinator C_i that has sent first a *Write Operation* message with the tagged value $(v_x, (r, c, 0))$ and later a *Write Operation* message with the tagged value $(v_y, (r, c, 0))$. Let us assume that $v_x \neq v_y$. This means that the value of the variable *PVal* managed by C_i has been modified between the two executions of line 16 of Task D. Yet, during that time interval, the tag $(r, c, 0)$ of C_i has not changed. Due to the test evaluated at line 15 , neither v_x nor v_y is equal to $\perp$.

The variable *PVal* can only be updated during Task C (either at line 8, at line 28, or at line 32 during the execution of line 10 of the code CAny) or during Task D (either at line 8, at line 11 during the execution of line 16 of the code DAny, or at line 14).

The coordinator C_i cannot update its variable *PVal* during an execution of line 8, line 28, or line 32 without increasing the second field of its tag. Due to Lemma 5, when the consensus number of C_i decreases, its round number increases. As the round number is a monotonically increasing variable and as the same tag appears in the two broadcast messages, we can conclude that the consensus number has not decreased and increased between the two broadcasts.

C_i can only execute line 8 at the end of the *Prepare* phase corresponding to the round number r. During this *Prepare* phase, it can not send any *Write Operation* messages. Due to Lemma 1 and due to the test of line 2, the tag used by C_i during two distinct *Propose* phases cannot be the same. Thus it is impossible that C_i send the tagged value $(v_x, (r, c), 0)$ during a *Propose* phase and the tagged value $(v_y, (r, c), 0)$ during another *Propose* phase (*i.e.*, after having executed at least one *Prepare* phase in between).

Finally, as $v_x \neq \perp$, C_j can update the variable *PVal* at line 10 only if another update of *PVal* has occurred before and has reset the value of *PVal* to $\perp$ without changing the tag. Consequently, a coordinator cannot broadcast *Write Operation* messages with different values but with the same tag: the lemma holds.

$\square_{Lemma\ 7}$

Lemma 8. *An acceptor cannot send two State messages that contain the same tag but different values.*

Proof

Let us consider that an acceptor A_i sends two *State* messages that contain different values. Between the two sending, A_i has modified at least once the value of its variable *VVal*. Due to Lemma 4, when an update occurs, the tag is necessarily changed and increases monotonically according to the $\prec$ order relation. Thus the Lemma holds.

$\square_{Lemma\ 8}$

Lemma 9. *When the tag of a coordinator is equal to $(r, c, 0)$, all the tagged values logged by this coordinator are equal.*

Proof The tag of a coordinator is logged in its variable *CTag*. The array *CVal* is used to log all the received tagged values that share this current tag. Let us consider two entries x and y such that $1 \leq x \leq n$ and $1 \leq y \leq n$. Assuming that the current tag of the coordinator is equal to $(r, c, 0)$, we have to show that, at any time, *SetCTag* $[x] \wedge$ *SetCTag* $[y] \Rightarrow$ *CVal*$[x]$ = *CVal*$[y]$. Note that this property always holds when the log is empty (*i.e.*, after the initialization phase and each time the log is reset to the empty set). Let us assume that, for the first time, the lemma does not hold at time t_2 for a given coordinator C_i. More precisely, at time t_2, the tag of C_i is equal to $(r, c, 0)$ and there exists two entries x and y such that *SetCTag* $[x] \wedge$ *SetCTag* $[y] \wedge$ (*CVal*$[x] \neq$ *CVal*$[y]$). Necessarily, there exists an instant $t_1 < t_2$ such that the log is empty at t_1 and the log is never reset to the empty set between t_1 and t_2. Obviously, during the time interval $[t_1, t_2]$, the tag of the coordinator C_i contained in its variable *CTag* remains the same. Indeed, each time the tag is modified, the log of tagged value is also reset. During the time interval, the entries x and y may have been updated several times. Yet, due to Lemma 8, an acceptor cannot send two different values with exactly the same tag $(r, c, 0)$. Thus, if the coordinator C_i updates an entry z, for the first time during $[t_1, t_2]$, when it receives a tagged value $(v_z, (r, c, 0))$ from A_z, any other *State* message sent (before or after) by A_z contains either the same value or a different tag. If the tag is the same, the same value v_z is assigned again to the entry z. Otherwise, depending on the result of the tests performed during Task C, the *State* message is either ignored or its processing leads to the reset of the log to the empty set. Consequently, between t_1 and t_2, entry x (respectively y) has been set at least once to the value v_x (respectively v_y) and then the value of this entry has never changed till t_2. As the set of logged values has never been reset to the empty set between t_1 and t_2, the tag $(r, c, 0)$ shared by all the logged values has also never changed between t_1 and t_2. The coordinator C_i has received at least one *State* message with a tagged value $(v_x, (r, c, 0))$ from an acceptor A_x and at least one *State* message with a tagged value $(v_y, (r, c, 0))$ from an acceptor A_y. Due to Lemma 3, none of these two value can be equal to $\perp$. Otherwise the tag $(r, c, 0)$ is equal to $(0, 0, 0)$ and the value $\perp$ cannot be logged in *CVal* due to the test performed at line 12 of Task C. If both v_x and v_y are different from $\perp$ and contained in *State* messages, Lemma 3 ensures that there exists a coordinator C_j (respectively C_k) that has previously broadcast a *Write Operation* message with the tagged value $(V_x, (r, c, 0))$ (respectively $(V_y, (r, c, 0))$). As the round number r is specific to a coordinator, we can conclude that $j = k$ and $r \bmod n = j$. Due to Lemma 7, a coordinator cannot broadcast two *Write Operation* messages with different values but the same tag: this contradicts the assumption that v_x and v_y are different and thus the lemma holds.

$\square_{Lemma\ 9}$

Lemma 10. *If two coordinators execute line 29 of Task C and decide respectively $< v_i, c >$ and $< v_j, c >$, then $v_i = v_j$.*

Proof

Let us assume that two coordinators, denoted C_{k_i} and C_{k_j}, decide respectively $< v_i, c >$ and $< v_j, c >$ when they execute line 29 of Task C. First we demonstrate that the value of the variable *CTag* managed by C_{k_i} (respectively, by C_{k_j}) is equal to $(r_i, c, 0)$ (respectively, $(r_j, c, 0)$) when Task C begins and is equal to $(r_i, c+1, 0)$ (respectively, $(r_j, c+1, 0)$) when the task ends. Indeed, as the log of tagged values contains enough values (at least two) when the condition of line 27 is evaluated, this log has not been reset to the empty set neither at line 9 nor at line 15 during the same execution of Task C. Consequently, the round number *CTag.Rnd* has not been changed during the execution of Task C. Also, the consensus number *CTag.Con*

has been updated only at line 30 of Task C. The third field of the tag was equal to 0 when the execution of the task starts (See line 26) and is still equal to 0 when the task ends (See line 30).

As a quorum did not exist during the previous execution of Task C, any coordinator that decides at line 29 has previously executed line 17 during the same task. Consequently, the decided values v_i and v_j are equal to the value of the variable *LVal* (See line 28) whose last update has been performed by the coordinator at line 17 of Task C. The *State* message received by C_{k_i} (respectively C_{k_j}) contains the tagged value $(v_i, (r_i, c, 0))$ (respectively $(v_j, (r_j, c, 0))$). Due to Lemma 5, the variable *CTag.Rnd* can only increase. Therefore, due to Lemma 3, the values v_i and v_j contained in the *State* messages are different from $\perp$. Otherwise the associated tag is equal to $(0, 0, 0)$ and the condition evaluated at line 12 of Task C is not satisfied: a contradiction. Due to Lemma 3, we can conclude that a *Write Operation* message that contains the tagged value $(v_i, (r_i, c, 0))$ has been sent by a leader C_i. A similar reasoning can be applied to the tagged value received by C_{k_j}: a *Write Operation* message that contains the tagged value $(v_j, (r_j, c, 0))$ has also been sent by a leader C_j. Consequently, the decided value v_i (respectively v_j) is equal to the value of the variable *PVal* managed by the leader C_i (respectively C_j) when it sends a *Write Operation* message at line 16 of Task D.

First, let us consider that $r_i = r_j$. In a *Write Operation* message, the round number *Op.Rnd* and the first field of the tag *Op.Tag.Rnd* are both corresponding to the value of the variable *RndLid* managed by the sender. As the possible values assigned to this variable are specific to a given coordinator, C_i and C_j are corresponding to the same coordinator denoted hereafter C_i. If $v_i \neq v_j$, the leader C_i has broadcast two *Write Operation* messages that contain the same tag but two different tagged values. This is in contradiction with Lemma 7.

Now, let us consider that $r_i \neq r_j$. Without loss of generality, let us assume that $r_i < r_j$. Note that it does not mean that $C_i \neq C_j$: a same coordinator can broadcast the two tagged values during two different *Propose* phases. Due to Lemma 9 and the fact that a majority quorum is observed by C_{k_i} (respectively by C_{k_j}) at line 27 of Task C, the decided value v_i (respectively v_j) has been received by the coordinator C_{k_i} (respectively C_{k_j}) from a majority of acceptors. We denote by Maj_{o_i} (respectively Maj_{o_j}) the majority quorum of acceptors observed by C_{k_i} (respectively C_{k_j}). Each tagged value has been accepted (at different times) by a majority of acceptors. We denote by Maj_{a_i} (respectively Maj_{a_j}) the set of acceptors that have adopted the tagged value $(v_i, (r_i, c, 0))$ (respectively $(v_j, (r_j, c, 0))$). By definition, $Maj_{o_i} \subseteq Maj_{a_i}$ and $Maj_{o_j} \subseteq Maj_{a_j}$.

First we demonstrate that no acceptor has adopted (and sent) a tagged value with a tag equal to $(r_i, c, 1)$. Let us assume that this scenario may happen. Due to Lemma 3, if a *State* message with such a tagged value is sent by an acceptor, a coordinator should have sent before a *Write Operation* message with a tagged value $(\top, (r_i, c, 0))$. This coordinator should be the leader of the round r_i, namely C_i. As mentioned before, C_i has sent another *Write Operation* message with the same tag and a value v_i. As v_i is decided by C_{k_i}, v_i is different from $\top$ (See line 27 of Task C). This is in contradiction with Lemma 7. Consequently, during the round r_i and the consensus instance c, an acceptor has adopted either no value or the tagged value $(v_i, (r_i, c, 0))$.

If we consider the set of acceptors Maj_{a_i} that have adopted the tagged value $(v_i, (r_i, c, 0))$, none will consider afterwards an *Operation* message with a tag (r', c', a') such that $(r', c', a') \prec (r_i, c, 0)$. Similarly, if we consider the set of acceptors Maj_{a_j} that have adopted the tagged value $(v_j, (r_j, c, 0))$, none have accepted before an *Operation* message with a tag (r', c', a') such that $(r_j, c, 0) \preceq (r', c', a')$. Consequently, we can focus on the coordinators that have broadcast a *Write Operation* message during a round period r_z such that $r_i \leq r_z \leq r_j$. By definition this set is finite: it contains at least two elements C_i and C_j and at most $r_j - r_i$ elements. Let m be the cardinality of this set and let us rename these coordinators C_{z_0}, C_{z_1}, $\cdots$, $C_{z_{m-1}}$. By definition, $C_{z_0} = C_i$ and $C_{z_{m-1}} = C_j$. For each coordinator C_{z_k} of this set, we consider the first *Write Operation* message sent by C_{z_k} during the *Propose* phase of the round period r_z. The notation $(vf_{z_k}, (r_{z_k}, cf_{z_k}, 0))$ is used to refer to the tagged value contained in this first *Write Operation* message. More generally, the notation $(vg_{z_k}, (r_{z_k}, cg_{z_k}, 0))$ is used to denote a tagged value contained in a *Write Operation* message broadcast by C_{z_k} during the round period r_{z_k}.

For any leader C_{z_k} and for any *Write Operation* message broadcast by this leader during the round period r_{z_k}, we demonstrate that the tagged value $(vg_{z_k}, (r_{z_k}, cg_{z_k}, 0))$ contained in the message satisfied the property $(cg_{z_k} > c) \vee ((cg_{z_k} = c) \wedge (vg_{z_k} = v_i))$. The proof is by induction. Let us consider the base case: $z_0 = i$. The coordinator C_i is executing the *Propose* Phase of the round period r_i when it sends for the first time a *Write Operation* message with the tagged value $(v_i, (r_i, c, 0))$. During the same round period, it can increase its consensus number but it can not decrease it. Indeed, due to Lemma 5, the consensus number contained in the variable *CTag.Con* managed by C_i can only decrease if the round number contained in its variable *CTag.Rnd* increases. If the value of the variable *CTag.Rnd* becomes strictly higher than r_i during an execution of Task C, the round number contained in the field *St.Rnd* of the received *State* message is also strictly higher than r_i due to Lemma 2. Therefore, at the end of Task C, the variable *Rnd* managed by C_i is also strictly higher than r_i. Consequently, if the consensus number of C_i decreases between two consecutive broadcasts of *Operation* messages, its current round periods ends and the round number of the next round period necessarily increases. During a round period executed by C_i, the value of the variable *PVal* can only change if the value of the variable *CTag.Con* increases. Consequently, while the value of the variable *CTag.Con* remains equal to c, the same value v_i is contained in all the the *Write Operation* messages

broadcast by C_i. Once C_i has broadcast at least one *Write Operation* message with the tagged value $(v_i, (r_i, c, 0))$, all the following *Write Operation* messages broadcast by C_i will satisfy the property $(cg_{z_0} > c) \vee ((cg_{z_0} = c) \wedge (vg_{z_0} = v_i))$.

Now let us assume that the property holds for all the value $z \leq z_{k-1}$. To demonstrate that the property $(cg_{z_k} > c) \vee ((cg_{z_k} = c) \wedge (vg_{z_k} = v_i))$ holds for all the *Write Operation* messages broadcast by C_{z_k}, we prove in a first step that $(cf_{z_k} > c) \vee ((cf_{z_k} = c) \wedge (vf_{z_k} = v_i))$. In other words, we demonstrate first that the property is satisfied by the first *Write Operation* message broadcast during the *Propose* phase. As C_{z_k} is not the first coordinator that sends a *Write Operation* message, we have $r_{z_k} > 1$. Consequently, C_{z_k} has necessarily executed a *Prepare* phase. While C_{z_k} is executing its *Prepare* phase, it broadcasts *Read Operation* messages that contain the round number r_{z_k} and, to terminate this phase, it must gather a majority of replies from a set of acceptors Maj_z. To be able to send later a *Write Operation* message with a tag $(r_{z_k}, cf_{z_k}, 0)$, C_{z_k} must only received during its *Prepare* phase *State* messages with a tag (r', c', a') such that $(r', c', a') \prec (r_{z_k}, cf_{z_k}, 0)$. At least one acceptor belongs both to Maj_{a_i} and Maj_z. If the tagged value of this acceptor is no more equal to $(v_i, (r_i, c, 0))$, the acceptor has adopted a tagged value provided either by C_{z_0}, C_{z_1}, ..., or $C_{z_{k-1}}$. Due to the induction assumption, in all the above cases, the value of the variable *VTag.Con* managed by the acceptor is greater or equal to c. Moreover, if the value of this variable is still equal to c, the associated value contained in the variable *VVal* is equal to v_i and the third field of the tag is still equal to 0. Consequently, the most recent tagged value received by C_{z_k} during its *Prepare* phase is a tagged value $(v', (r_{highest}, c', a'))$ such that $r_i \leq r_{highest} < r_{z_k}$ and $(c' > c) \vee ((c' = c) \wedge (v' = v_i) \wedge (a = 0))$.

Obviously, in its first *Write Operation* message that contains the tagged value $(vf_{z_k}, (r_{z_k}, cf_{z_k}, 0))$, the consensus number is such that $cf_{z_k} \geq c' \geq c$. Indeed, at the end of the *Prepare* phase, the value of the variable *CTag.Con* managed by C_{z_k} is equal either to c' or to $c' + 1$. The last case may occurs if enough tagged values associated to the highest tag ever observed have been gathered to take a decision related to the consensus instance c' before the end of the *Prepare* phase. If we consider the case where $cf_{z_k} = c' = c$, all the tagged values $(v_i, (r_{highest}, c, 0))$ received during the *Prepare* have been logged and the log of tagged value managed by the coordinator C_{z_k} is not empty. The coordinator C_{z_k} has received some (at least one) *State* message with the tagged value $(v_i, (r_{highest}, c, 0))$ during its *Prepare* phase. After the first receipt of such a tagged value, the statement *Reset(SetCTag,false)* has never been executed (at least till the *Prepare* phase ends). Indeed, by assumption, no decision related to the consensus instance c has been taken and no highest tag has been observed. Consequently, at the end of its *Prepare* phase, when C_{z_k} executes line 7 of Task D, the corresponding condition is evaluated to true (*i.e.*, the log of tagged values is not empty). The value v_i is selected by the leader when it executes line 8 of Task D. As the third field of the tag is necessarily equal to 0, any value selected in the log is equal to v_i. As v_i is different from $\perp$, the evaluation of the test of line 14 returns false while the evaluation of the test of line 15 returns true. Consequently, C_{z_k} will send a first *Write Operation* message with a tagged value $(v_i, (r_{z_k}, c, 0))$. Due to Lemma 7, C_{z_k} will broadcast the same tagged value while its tag remains equal to $(r_{z_k}, c, 0)$. As v_i is different from $\top$, the third field of the tag remains equal to 0. Following the demonstration done in the base case for C_i, when the leader C_{z_k} decreases its consensus number, it stops its current round period numbered r_{z_k}. Consequently, the consensus number stored in the variable *CTag.Con* can only increase during the round period r_{z_k}. Again, while this value remains equal to c, the value of the variable *PVal* remains equal to v_i. Consequently, the property $(cg_{z_k} > c) \vee ((cg_{z_k} = c) \wedge (vg_{z_k} = v_i))$ holds for any *Write Operation* message broadcast by C_{z_k} during the round period r_{z_k}.

If the leader $C_{z_m} = C_j$ broadcast a *Write Operation* message with the tagged value $(v_j, (r_j, c, 0)$, we can conclude that $v_j = v_i$. The Lemma holds.

$\square_{Lemma\ 10}$

Lemma 11. *If two coordinators execute line 12 of Task CAny and decide respectively* $< v_i, c >$ *and* $< v_j, c >$*, then* $v_i = v_j$*.*

The proof of Lemma 11 follows the structure adopted during the proof of Lemma 10. To decide a value, a coordination must gather a larger quorum of tagged values that share the same tag. All the logged values are not necessarily equal. But each acceptor can provide at most one direct value. The function *CollisionSafe* called at line 8 of the code CAny is used to prevent a deadlock that may occurs when the values provided by the proposers are different. A simple implementation of this may return false if two different values have been proposed. The selection of the most frequent value among the set of logged values is done when a coordinator decides (code CAny) or when a leader ends its prepare phase and try to determine if there exists a value already proposed by another coordinator during the same consensus instance. When a decision has been reached, the next *prepare* phases select always the decision value.

Lemma 12. *If two coordinators execute line 9 of Task C and decide respectively* $< v_i, c >$ *and* $< v_j, c >$*, then* $v_i = v_j$*.*

The proof of Lemma 12 relies on the fact that the variables *CTag.Con* and *Dval* are strongly related and always updated simultaneously (even when the consensus number decreases).

Lemma 13. *If two coordinators execute line 4 of Task C and decide respectively* $< v_i, c >$ *and* $< v_j, c >$*, then* $v_i = v_j$*.*

The proof of Lemma 13 relies on the fact that an entry k in the log managed by an acceptor can either contained the value $\perp$ or the computed decision value corresponding to the consensus instance k. Therefore, the proof is based on the three previous lemmas.

Lemma 14. *If a coordinator C_i decides $< v, c >$ then at least one coordinator C_j has previously executed DecidePush($< v, c >$) at line 29 of Task C or at line 12 of the code CAny..*

This Lemma distinguishes the computed decision from the adopted one.

Theorem 15 (Validity-Non-triviality)**.** *If a process executes DecidePush($< v, c >$), then a process has previously executed ProposePull(c) and obtained either the initial value v or the special mark $\top$.*

Proof Let C_i be a coordinator that decides $< v, c >$. According to Lemma 14, there exists a coordinator C_x that has also decided $< v, c >$ at line 21 of Task C or at line 12 of the code CAny. To obtain the decision value v, C_x has selected a tagged value $(v, (r, c, a))$ among its set of logged tagged values. The selected tagged value was previously contained in a *State* message received from an acceptor. Due to Lemma 3, a coordinator C_y has previously broadcast a *Write Operation* message that contains the tagged value $(\top, (r, c, 0))$ or the tagged value $(v, (r, c, 0))$. Necessarily, v is different from $\perp$ (See line 15). The sent value v is contained in the variable *PVal*. This variable is initialized to $\perp$. Thus, as $v \neq \perp$, this variable has been updated since the initialization phase. We can ignore the updates of this variable that assign the value $\perp$ to the variable *PVal*. As $v \neq \perp$, the last update has been done either at line 8, at lien 11 or at line 14. Obviously, in the last case, the theorem is satisfied: C_y has called the *ProposePull* function during the consensus instance c and obtained either an initial value or $\top$.

In the two other cases, to define the new value of *PVal* , C_y selects the value v in its log of tagged value. This is done just before the end of the *Prepare* phase. Let us assume that the selected tagged value is $(v, (r', c, a))$. During the following *Propose* phase corresponding to round r, C_y will broadcast a *Write Operation* message that contains the tagged value $(v, (r, c, a))$. Due to Lemma 1, we have necessarily $r' < r$ and as the log was not empty, we have necessarily $r' > 1$. Again, both the value v and the tag (r', c, a), were contained in a message sent by an acceptor. So the same reasoning can be applied recursively. As by definition, the round number is always greater or equal to 1, we can conclude that the theorem holds in all the cases. $\square_{Theorem\ 15}$

Theorem 16 (Validity-Atomicity)**.** *If a process executes ProposePull(c) with $c > 1$ then it has previously executed DecidePush($< v, c - 1 >$).*

Proof The function *ProposePull* is called at line 14. If the variable *CTag.Con* is equal to c at that time, due to Lemma 6 we know that either $c = 1$ or a decision related to the consensus instance $c - 1$ has been made before. $\square_{Theorem\ 16}$

Theorem 17 (Validity-Unicity)**.** *If a process executes several times ProposePull(c) then only the last call may return a value different from the special mark $\perp$.*

The function *ProposePull* is called at line 14 only if the current value of the variable *PVal* is equal to $\perp$. To prove that the theorem is true, we show that each time the variable *PVal* is reset to the value $\perp$, the consensus number contained in the variable *VConTag* increases.

Theorem 18 (Agreement)**.** *If two coordinators C_i and C_j decides respectively $< v_a, c >$ and $< v_b, c >$ then $v_a = v_b$.*

The proof is based mainly on Lemmas 10, 11, 12, 13 and 14.

Theorem 19 (Termination-Progress)**.** *If any correct process can execute ProposePull(c) and eventually obtain an initial value, then at least one process eventually executes DecidePush($< v, c >$).*

Proof

We assume that enough acceptors are non-faulty (at least a majority). We also assume that at least one non-faulty proposer is able to propose a value to the successive leaders (and direct values if the proposer provides first an $\top$ value) and at least one non-faulty learner is reading to learn the decision value. Every Paxos-like protocol assumes a *leader election* module that is required to (eventually and for a sufficiently long time) provide a unique and non-faulty leader. This requirement is needed to guarantee liveness properties.

In the proposed solution, a leader is a non-faulty coordinator that is elected by a large-enough quorum of acceptors. Each coordinator periodically receives *State* messages from the acceptors. Such a message informs the coordinator about the fact that the sender supports it (or not) as leader. This *leader election* algorithm ensures that eventually, a unique coordinator will gather enough information to become the unique leader, for sufficiently long time to succeed in reading and writing, at a majority of acceptors. More precisely, we assume that each acceptor in a majority quorum Q eventually elects a coordinator C_i as a leader, after querying their local *leader election* module. C_i discovers it is the new leader after gathering support from a quorum of acceptors (when the test at line 1 becomes true). If the leader has not yet received the highest values for the cycle and consensus numbers, it will eventually receive these fresh values from the acceptors. Indeed, due to the

retransmission mechanism implemented by Task B, each acceptor periodically sends its current information to the leader. Hence, the leader will gather the most up-to-date information.

If no process decides before, the selected leader C_i will eventually be a correct process and it will be guaranteed by the *leader election* module to be the unique leader for sufficiently long time. In that case, it will succeed in reading at a majority of acceptors and then writing either a new proposal or a previously written value. Indeed C_i can not be blocked during the *Prepare* phase. In the worst case, this *Prepare* phase will last until C_i sends a *Read* message with the highest round number ever proposed by a leader. C_i will select a value to propose at the end of the *Prepare* phase (See line 8) or it will obtain it from a proposer when it calls the *ProposePull* function (See line 14). The *Write Operation* will succeed at a quorum of acceptors. In the case of an Any value $\top$, collisions are detected and the leader will start a new round period with the more frequent direct value it has observed. Indeed, the fact that the round number r of C_i is the highest ever generated is sufficient to ensure that each acceptor will adopt the first value proposed by C_i and tagged with $(r, c, 0)$ whatever the value of c. C_i can not ensured that the value c of its variable *VConTag* is the highest consensus number ever generated and sent in a *Write Operation* message. But, due to the test performed at lines 1-2, an acceptor may accept the first value it received with the tag $(r, c, 0)$ even if its consensus number is currently equal to $c + 1$ and will regress to c. If a majority of acceptors adopts the proposed tagged value, C_i eventually observes a majority quorum and decides *DVal* at line 28.

$\square_{Theorem\ 19}$

Theorem 20 (Termination-Persistence)**.** *If a process executes DecidePush*$(< v, c >)$ *then at least one correct process eventually executes DecidePush*$(< v, c >)$

Proof a completer First let us assume that the consensus instance c is the last one. If the first leader that decides $(< v, c >)$ is correct, the lemma holds. Otherwise, it will crash and a new leader will be chosen. This new leader will execute a *Prepare* phase with a higher round number. In the worth case, all the faulty process will act as a leader and crash before a first correct process C_i acts as a new leader. Whatever the number of faulty processes that may have made a decision during the consensus instance c before crashing, the correct leader C_i will obtain the last consensus number and all the previously decided values thanks to the *RetrieveDec* function. $\square_{Theorem\ 20}$

8 Conclusion

The Paxos-MIC protocol follows the approach of FastPaxos [1] which allows a leader to execute several consensus instances during the same *Propose* phase. This mechanism is extended with the possibility of executing an *Any* round, like in Fast Paxos [9], thus reducing the latency to two communication steps, in favorable circumstances. Paxos-MIC also guarantees the persistence of all the previous decisions before a new one is made. As a direction for future works, we plan to conduct simulations and experimentations to observe how favorable circumstances occur and to obtain a thorough analysis of the protocol's behavior.

A slight modification of the semantic of an *Any* value may transform the proposed protocol into an abstraction of Multi Writer Multi Reader atomic register. Let us assume that any initial value v provided by a proposer corresponds to a *Write*(v) request and any value $\top$ corresponds to a *Read* request. In that case, an *Any* value is no more used to bypass the coordinator. Instead, if rather than waiting for a direct value provided by a proposer, an acceptor replaces immediately the $\top$ value it receives by its current value, the constructed sequence of decision values corresponds to a decided interleaving of read and write values. Additional transformations of the protocols are required, for example, to record the identity of the process associated to an invocation.

References

[1] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. *ACM SIGACT News*, 34(1):47–67, March 2003.

[2] R. Boichat, P. Dutta, and R. Guerraoui. Asynchronous leasing. *IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[4] G. Chockler, R. Guerraoui, and I. Keidar. Amnesic distributed storage. *Proc. of the 21st International Symposium on Distributed Computing (DISC'07)*, pages 139–151, 2007.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[6] M. Hurfin, J.P. Le Narzul, F. Majorczyk, L. M, A. Saidane, E. Totel, and F. Tronel. A dependable intrusion detection architecture based on agreement services. *Proc. of the 8th Int. Symposium on Stabilization Safety and Security*, pages 378–394, November 2006.

[7] L. Lamport. The part-time parliament. *ACM Transaction on Computer Systems*, 16(2):133–169, May 1998.

[8] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, December 2001.

[9] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[10] B. Lampson. The abcds of paxos. *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing*, 2001.

[11] J.P. Le Narzul and M. Hurfin. Design and performance evaluation of a resource allocation system based on agreement services. *Proc. of the 10th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 387–393, Sept 2008.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[13] J. P. Martin and L. Alvisi. Fast byzantine consensus. *Proc. of the Int. Conference on Dependable Systems and Networks*, pages 402–411, June 2005.

[14] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[15] G. M. D. Vieira and L. E. Buzato. On the coordinator's rule for fast paxos. *Information. Processing Letters*, pages 183–187, March 2008.